感受设计演变过程中所蕴含的大智慧，
体会乐与怒的程序人生中值得回味的一幕幕。

大话设计模式

小菜

大鸟

对酒当歌
人生几何

设计模式的趣味解读，面向对象的深入剖析。
在诙谐与温馨中做一次面向对象编程思维的体操。

程杰 著

清华大学出版社



原创经典

VISIT…

LANZAROTE Caliente.com

感受设计演变过程中所蕴含的大智慧，

体会乐与怒的程序人生中值得回味的一幕幕。

大话设计模式



设计模式的趣味解读，面向对象的深入剖析。

在诙谐与温馨中做一次面向对象编程思维的体操。

程杰 著

清华大学出版社



原创经典

大话设计模式

程 杰 著

清 华 大 学 出 版 社
北 京

目录

内容简介
序
本书出版后的读者评论
前言
第1章 代码无错就是优？——简单工厂模式
1.1 面试受挫
1.2 初学者代码毛病
1.3 代码规范
1.4 面向对象编程
1.5 活字印刷，面向对象
1.6 面向对象的好处
1.7 复制vs.复用
1.8 业务的封装
1.9 紧耦合vs.松耦合
1.10 简单工厂模式
1.11 UML类图
第2章 商场促销——策略模式
2.1 商场收银软件
2.2 增加打折
2.3 简单工厂实现
2.4 策略模式
2.5 策略模式实现
2.6 策略与简单工厂结合
2.7 策略模式解析
第3章 拍摄UFO——单一职责原则
3.1 新手机
3.2 拍摄
3.3 没用的东西
3.4 单一职责原则
3.5 方块游戏的设计
3.6 手机职责过多吗？
第4章 考研求职两不误——开放-封闭原则
4.1 考研失败
4.2 开放-封闭原则
4.3 何时应对变化
4.4 两手准备，并全力以赴
第5章 会修电脑不会修收音机？——依赖倒转原则
5.1 MM请求修电脑
5.2 电话遥控修电脑
5.3 依赖倒转原则
5.4 里氏代换原则

5.5 修收音机
第6章 穿什么有这么重要？——装饰模式
6.1 穿什么有这么重要？
6.2 小菜扮靓第一版
6.3 小菜扮靓第二版
6.4 装饰模式
6.5 小菜扮靓第三版
6.6 装饰模式总结
第7章 为别人做嫁衣——代理模式
7.1 为别人做嫁衣！
7.2 没有代理的代码
7.3 只有代理的代码
7.4 符合实际的代码
7.5 代理模式
7.6 代理模式应用
7.7 秀才让小六代其求婚
第8章 雷锋依然在人间——工厂方法模式
8.1 再现活雷锋
8.2 简单工厂模式实现
8.3 工厂方法模式实现
8.4 简单工厂vs.工厂方法
8.5 雷锋工厂
第9章 简历复印——原型模式
9.1 夸张的简历
9.2 简历代码初步实现
9.3 原型模式
9.4 简历的原型实现
9.5 浅复制与深复制
9.6 简历的深复制实现
9.7 复制简历vs.手写求职信
第10章 考题抄错会做也白搭——模板方法模式
10.1 选择题不会做，蒙呗！
10.2 重复＝易错＋难改
10.3 提炼代码
10.4 模板方法模式
10.5 模板方法模式特点
10.6 主观题，看你怎么蒙
第11章 无熟人难办事？——迪米特法则
11.1 第一天上班
11.2 无熟人难办事
11.3 迪米特法则
第12章 牛市股票还会亏钱？——外观模式
12.1 牛市股票还会亏钱？
12.2 股民炒股代码

12.3 投资基金代码
12.4 外观模式
12.5 何时使用外观模式
第13章 好菜每回味不同——建造者模式
13.1 炒面没放盐
13.2 建造小人一
13.3 建造小人二
13.4 建造者模式
13.5 建造者模式解析
13.6 建造者模式基本代码
第14章 老板回来，我不知道——观察者模式
14.1 老板回来？我不知道！
14.2 双向耦合的代码
14.3 解耦实践一
14.4 解耦实践二
14.5 观察者模式
14.6 观察者模式特点
14.7 观察者模式的不足
14.8 事件委托实现
14.9 事件委托说明
14.10 石守吉失手机后的委托
第15章 就不能不换DB吗？——抽象工厂模式
15.1 就不能不换DB吗？
15.2 最基本的数据访问程序
15.3 用了工厂方法模式的数据访问程序
15.4 用了抽象工厂模式的数据访问程序
15.5 抽象工厂模式
15.6 抽象工厂模式的优点与缺点
15.7 用简单工厂来改进抽象工厂
15.8 用反射＋抽象工厂的数据访问程序
15.9 用反射＋配置文件实现数据访问程序
15.10 无痴迷，不成功
第16章 无尽加班何时休——状态模式
16.1 加班，又是加班！
16.2 工作状态-函数版
16.3 工作状态-分类版
16.4 方法过长是坏味道
16.5 状态模式
16.6 状态模式好处与用处
16.7 工作状态-状态模式版
第17章 在NBA我需要翻译——适配器模式
17.1 在NBA我需要翻译！
17.2 适配器模式
17.3 何时使用适配器模式

17.4 篮球翻译适配器
17.5 适配器模式的.NET应用
17.6 扁鹊的医术
第18章 如果再回到从前——备忘录模式
18.1 如果再给我一次机会……
18.2 游戏存进度
18.3 备忘录模式
18.4 备忘录模式基本代码
18.5 游戏进度备忘
第19章 分公司＝一部门——组合模式
19.1 分公司不就是一部门吗？
19.2 组合模式
19.3 透明方式与安全方式
19.4 何时使用组合模式
19.5 公司管理系统
19.6 组合模式好处
第20章 想走？可以！先买票——迭代器模式
20.1 乘车买票，不管你是谁！
20.2 迭代器模式
20.3 迭代器实现
20.4 .NET的迭代器实现
20.5 迭代高手
第21章 有些类也需计划生育——单例模式
21.1 类也需要计划生育
21.2 判断对象是否是null
21.3 生还是不生是自己的责任
21.4 单例模式
21.5 多线程时的单例
21.6 双重锁定
21.7 静态初始化
第22章 手机软件何时统一——桥接模式
22.1 凭什么你的游戏我不能玩
22.2 紧耦合的程序演化
22.3 合成/聚合复用原则
22.4 松耦合的程序
22.5 桥接模式
22.6 桥接模式基本代码
22.7 我要开发“好”游戏
第23章 烤羊肉串引来的思考——命令模式
23.1 吃烤羊肉串！
23.2 烧烤摊vs.烧烤店
23.3 紧耦合设计
23.4 松耦合设计
23.5 松耦合后

23.6 命令模式
23.7 命令模式作用
第24章 加薪非要老总批？——职责链模式
24.1 老板，我要加薪！
24.2 加薪代码初步
24.3 职责链模式
24.4 职责链的好处
24.5 加薪代码重构
24.6 加薪成功
第25章 世界需要和平——中介者模式
25.1 世界需要和平！
25.2 中介者模式
25.3 安理会做中介
25.4 中介者模式优缺点
第26章 项目多也别傻做——享元模式
26.1 项目多也别傻做！
26.2 享元模式
26.3 网站共享代码
26.4 内部状态与外部状态
26.5 享元模式应用
第27章 其实你不懂老板的心——解释器模式
27.1 其实你不懂老板的心
27.2 解释器模式
27.3 解释器模式好处
27.4 音乐解释器
27.5 音乐解释器实现
27.6 料事如神
第28章 男人和女人——访问者模式
28.1 男人和女人！
28.2 最简单的编程实现
28.3 简单的面向对象实现
28.4 用了模式的实现
28.5 访问者模式
28.6 访问者模式基本代码
28.7 比上不足，比下有余
第29章 OOTV杯超级模式大赛——模式总结
29.1 演讲任务
29.2 报名参赛
29.3 超模大赛开幕式
29.4 创建型模式比赛
29.5 结构型模式比赛
29.6 行为型模式一组比赛
29.7 行为型模式二组比赛
29.8 决赛

29.9 梦醒时分
29.10 没有结束的结尾
附录A 培训实习生——面向对象基础
A.1 培训实习生
A.2 类与实例
A.3 构造方法
A.4 方法重载
A.5 属性与修饰符
A.6 封装
A.7 继承
A.8 多态
A.9 重构
A.10 抽象类
A.11 接口
A.12 集合
A.13 泛型
A.14 委托与事件
A.15 客套
附录B 参考文献

图书在版编目（**CIP**）数据

大话设计模式/程杰著.-北京：清华大学出版社，2007.12

ISBN 978-7-302-16206-3

I.大… II.程… III.软件设计　IV.TP311.5

中国版本图书馆CIP数据核字（2007）第152713号

责任编辑：陈　冰

责任校对：张　剑

责任印制：

出版发行：清华大学出版社　　地　　址：北京清华大学学研大厦A座

http://www.tup.com.cn　　邮　　编：100084

c-service@tup.tsinghua.edu.cn

社总机：010-62770175　　邮购热线：010-62786544

投稿咨询：010-62772015　　客户服务：010-62776969

印刷者：

装订者：

经　　销：全国新华书店

开　　本：203×260　印张：24.5　　字数：683千字

版　　次：2007年12月第1版　　印次：2007年12月第1次印刷

印　　数：1～000

定　　价：　　元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：010-62770177转3103　产品编号：026653-01

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：**010-62782989**

13501256678 13801310933

本书是一本程序集？**NO**！

本书是一本故事集？**NO**！

本书是一本通过故事讲述程序如何设计的方法集。了解优秀软件设计的演变过程比学习优秀设计本身更有价值，因为设计的演变过程中蕴藏着大智慧。

感受设计演变过程中所蕴含的大智慧，

体会乐与怒的程序人生中值得回味的一幕幕。

设计模式的趣味解读，面向对象的深入剖析。

在诙谐与温馨中做一次面向对象编程思维的体操。

内容简介

本书通篇都是以情景对话的形式，用多个小故事或编程示例来组织讲解GoF（设计模式的经典名著——Design Patterns: Elements of Reusable Object-Oriented Software，中译本名为《设计模式——可复用面向对象软件的基础》的四位作者Erich Gamma、Richard Helm、Ralph Johnson，以及John Vlissides，这四人常被称为Gang of Four，即四人组，简称GoF）总结的23个设计模式。本书共分为29章。其中，第1、3、4、5章着重讲解了面向对象的意义、好处以及几个重要的设计原则；第2章，以及第6到第28章详细讲解了23个设计模式；第29章是对设计模式的全面总结。附录部分是通过一个例子的演变为初学者介绍了面向对象的基本概念。本书的特色是通过小菜与大鸟的趣味问答，在讲解程序的不断重构和演变过程中，把设计模式的学习门槛降低，让初学者可以更加容易地理解——为什么这样设计才是好的？是怎样想到这样设计的？以达到不但授之以“鱼”，还授之以“渔”的目的。引导读者体会设计演变过程中蕴藏的大智慧。

本书适合编程初学者或希望在面向对象编程上有所提高的开发人员阅读。

序

这本书最初起源于作者程杰在其博客中所写的连载文章——《小菜编程成长记》。随着文章的一篇篇发布，这些文章新颖的表现形式和独特的风格受到了众多读者的关注和喜爱，很多人在博客中留下了评语。有些虽然只有短短的一句话，但也可以看出是对作者由衷的感谢。作为本书的策划编辑，最初我也是在博客园中浏览博文时阅读到这些文章的，我的直觉和网友们热情洋溢的评语告诉我，这些文章有作为一部书出版的价值，于是我就联系了程杰。几个月后，我拿到了这部书的初稿。

初审后，我发现书稿中存在一些问题。比如，当时书稿中还没有对UML类图进行讲解的内容，这会导致初学者学习后面的内容时感到理解困难，于是我请作者在第1章中增加了UML类图这一节，这是简洁却精彩的一节；另外，当时作者为了便于表达某些举例的含义，有相当数量的代码都是用中文编写的，虽然中文代码看似易懂，但却会令绝大多数早已熟悉了英文代码的程序员们感到困惑和难以阅读，所以我请作者把代码改回为程序员们所熟悉的英文代码，并同时添加了更详细的中文注释。经过几番认真和辛苦的修改与调整，现在，这本书在你的手中了。

对于这本书，我想说的是，其中的很多篇章非常的精彩，会令你禁不住叫好，但也有一些篇章会显得有些拖沓，或者是有些牵强，然而，随着你读过那些精彩的段落，读过那些不那么精彩的段落，最终，你会读到书的最后一页（很多书不能使你做到这一点），当你读完全书时，你会发现，你的心情很愉快，很平静，即使是那些当时看起来不那么精彩的段落，现在也都成为了这温馨故事的一部分。你会记得书中那个好学、天真、而又执著的小菜，也会记得那个善于启发，经验老道的大鸟。

下面这些是来自作者博客的网友评论，看完这些热情洋溢的评论，就和作者一起，进入设计模式的大话境界吧。

本书策划编辑　陈冰

2007年10月18日

网友评论

daigua：看到这篇精彩的成长记，我连饭都不想吃了，什么事都不想做，就想把它看完。写得太好了！是啊，现在很多教材都太枯燥了，不好理解。其实书的意义就在于让人学到知识，而不在于用什么方式，为什么一定要那么教条呢，只要能让人比较容易地学到书里的知识就是一本好书。谢谢你啊，给了我很大的信心。我现在很有信心把编程进行到底，哈哈。

光头小松鼠：绝对经典！一篇小故事，把程序的灵活性、可扩展性、可维护、可复用等说得怎一个妙字了得！

沉默天蝎：感激，让我这个菜鸟顿悟。这样的写法太好了，如果老大你出书，我肯定购买！

碳碳：这种学习的方式真的很神奇，尽管每个人都能想到，但不是每个人都能做到。或许可以把系列文章归档出书，说不定会收到追捧，呵呵。

Bryant：真的是太棒了！我原来看过一些有关设计模式的书，都觉得太抽象，根本就不能理解，也不知道啥时候能用上。看过你写的这些文章，才知道了应该怎样在实际中运用这些模式，而且文笔非常的幽默，享受！Thx ^_^ 支持！有个建议，最好慢慢地把所有的设计模式都聊聊！

Bryant：不错，楼主说的非常幽默，通俗，把我们一步一步带入面向对象的世界thx ^_^

Bryant：太棒了，我正是这样初学设计模式的小菜，需要这样的文章，谢谢楼主！

菜鸟飞：楼主，加油，支持你。在这里献上崇高的敬意，不管你有没有感受到我挚热的目光。请你相信，有这样一些人一直在默默地关注着你，期待着你。

wdx2008：非常好！！！幽默，搞笑，易懂，真神人也，鬼神不可测！支持楼主！！

空明流转：呵呵，楼主说得蛮好。国外的文章好就好在有例子，“废话”多，所以比较好理解。至于行文风格嘛，这个倒是因人而异的。我个人就偏向于论文式的行文风格，逻辑严密，层层递进，阐述也很清晰。就有点像有序数组，二分法就能轻松查找到自己想要的东西，但国内的那种论文式的文章，呵呵，我看是卖弄的成分居多，实作的成分偏少，所以才那么难读的吧。

Char：现在的大学就缺少这种既通俗易懂，又有内容的东西。

Apple：不错，学习了。希望博主能再接再厉多写点，看了很多书都没有看你的文章明白得快。

SnowDoggie：呵呵，挺好的。其实要想找个绝对没有漏洞的例子是很辛苦的，关键在于文章本身能说明问题，能体现作者的意图就足够了。昨天和朋友一起爬山的时候还讨论了你的文章风格，其实最有用的还是你这种寓教于乐，步步深入的风格，阳春白雪的经典虽然是经典，大众却不见得喜欢。

Jerry：不错的文章，简单明了，又不乏趣味，好的文章就得顶下。

izhizhe2000：很好，整个系列写完之后可以出书了，保证受大学生的广泛欢迎！

mekong：很是欣赏这样幽默风趣又不失睿智深刻的文字。

Wuyisky：呵呵，楼主不仅程序写得好，而且还有文学天赋。佩服！

Jack：真正的高手是用最生动的语言，最简单的例子，这才是真正的“深入浅出”。赞！！！老兄，加油，继续哟。

BoyLee：人才，爱死你了。做了一年外包，没技术含量。正打算从头学习这些东西，这样的方式我最喜欢了。

Leoxu：很不错，对正在找工作的我有很大的帮助。以后会多来光顾。

Ame：写得承上启下，始终有一主干线贯穿，作者的文字功底很强啊！

Artech：我很喜欢你的写作风格！以一种调侃的方式讲明一个深奥的问题。我一直在尝试如何以一种让每个人都懂得的语言来向大家分享我所理解的.NET。你给了我一个启发。

8：醍醐灌顶！感谢，领悟了不少东西！！！

Yufengly：真是太容易理解了，而且看后印象深刻，继续努力！期待下文……支持作者！

Sopper：支持，例子举得很形象，写得很棒，以后会常来关注。

d：会技术的高人有很多，但能把技术讲得如此通俗易懂的高人并不多，你是一个，谢谢～～～

white.wu：非常喜欢您这种授人以“渔”的文章。

Answer：强啊，本菜鸟受益很大，谢谢。

Hanlei：强，很受益啊，感谢楼主，写出这么好的文章来。

金色海洋（**jyk**）：继续呀，我们期待中……，写得很好，一看就懂。

DSharp：看博客园这么久了，终于看到一篇有中国特色的好文。

本书出版后的读者评论

当当网的本书读者评论（当当网是只有买了书的读者才能发表评论的）

zenmelpengyou：这本书是我从识字以来看过的最好的一本书了，诙谐幽默而又能学到知识~~~~~~~

aivdesign：正在看。真是受益匪浅，作者不但用很通俗易懂的方式介绍了设计模式，同时还渗透了求职、面试、办公室文化等很多生活场景，让设计模式的概念融入其中，真是想忘也不容易忘啊，呵呵！

wxhwzz：书很不错~诙谐幽默！应该是一种很好的创新，绝对是设计模式入门级的好书，赞一个！当当发货的速度也很快，两天就收到了！很感动~~

hwj_no1：确实是本好书，在入门阶段和晋级阶段，都可以看看。并且能看到不同的启发，确实适合我这样的初级选手，语言和风格都很清新，在初级的不同阶段阅读，有横看成岭侧成峰的感觉。

asky99：适合设计模式入门的初学者研读，技术要点融汇在诙谐幽默的小说当中，是我读过的最轻松的技术类书籍，支持作者。博客园的朋友。

蓝天**DotNet**：刚看了几天，感觉很不错的一本书。不象其它设计模式书籍，都是纯理论的，刚开始看就叫人难以捉摸。

Sofut：这个是我在当当买过的最好的一本书！

xpwang_leo：此书乃不可多得之佳作，通俗易懂，读完很容易吸收，其实以前也读过很多设计模式的书籍，发现再重头来读这本书，有种醍醐灌顶的感觉，以前生涩难懂的地方，现在豁然开朗，如果想学设计模式，读它吧。

niyo：不错，浅显易懂，看小说一样。

bellxie：以前每次看模式的书，总是看到一半就昏昏欲睡，这本书一口气全部看完了。内容深入浅出，通俗易懂。总算看懂了几个模式。不错的好书。

苗苗老师：看惯了Wrox的书，这次换换思维方式。看中国人写的书就是容易接受，毕竟是用母语思维然后用母语写的，不经过翻译。看书

的过程不像看外国人写的书，总是走走停停，需要回味一下。看这本书是行云流水，一气呵成。

siso：单单看后面的基础知识，就比许多C#教程还好。

adamzhang：这本书我每天花一个小时左右来读，差不多两三周就看完了。非常通俗易读，而且讲得很好，能联系现实生活，很有趣味性。

卓越亚马逊的本书读者评论

ceduce193：由于以前就一直关注作者的《小菜编程成长记》，发现china-pub上有，就在那儿买了。刚从china-pub拿到书，大致翻了一下，感觉很不错。正如作者在前言所讲的，“精彩的代码是如何想出来的，要比看到精彩的代码更加令人期待。”每一个设计模式都是从小故事入手，然后一步一步在小菜犯错大鸟启发中引出设计模式。书中有一段讲解这样写，“为了回归的大局，增加一种制度又何尝不可，一个国家，两种制度，这在政治上，是伟大的发明。在软件设计模式中，这种不能修改，但可以扩展的思想也是最重要的一种设计原则，它就是开放-封闭原则”，这个例子举得太精彩了。反而觉得作者公开的两章不能完全代表书的全貌，强烈推荐购买。

stoat：此书不错，比晦涩的教材好理解多了。通过示例代码的不断完善引出了各个设计模式的特征和使用方法，还有风趣的人物对话。我买的时候38块多，才过几天就降了5快！！郁闷啊！卓越的物价变得太快了吧！

互动出版网的本书读者评论

hermitbab：我觉得不错，至少原先看的那些设计模式都很枯燥，看这个书就好像看小说一样，看好了记忆也深刻。

tiananz2006：在书店看了两章，整书的思路就象Head First的思路，当然不如Head First所举的范例那么有震撼力，但仍然是本入门的好书！购买此书的理由主要是：1·Head First在网上都有下载了，可以对比着读；2·所谈范例更针对于学生学习编程；3·价格便宜，支持国内原创。

songlonglong：感觉很不错，四人团的书太晦涩，如果和这本书结合看，效果很好。

第二书店的本书读者评论

c25894670：我对于设计模式只是一知半解，一直感觉GoF大师的作品以自己的水平实在无法理解。这本书实在是非常适合我这种不喜欢自己动脑子理解大师语言的人，用通俗的例子让我理解设计模式的精

髓，我感觉爽得要死……推荐想入门设计模式的人看这本书，入门的不二法门……

lizhizhe2000：在博客上看过作者的一些连载，写得很风趣，很适合初学者看看，对那些工作了两三年的人也有实际价值。

124.114.129.*：支持这样的书籍，大道至简至易。

125.35.4.*：刚收到书，看了几页，感觉这本书应该是程序员学习的第一本书，也许在讲设计模式的书中这不是最好的。但肯定是能让你看了不困能从头到尾坚持看完的，而且每一种设计模式的用法只用一句话就简明概括出来了，比起以前买的看了二、三十页就头脑发胀混混欲睡的牛人的著作，这本书真的可以称得上是《设计模式——菜鸟天书》，强烈推荐。到家才37.2，便宜透了。

58.246.92.*：这本书完全是作者的工作经验的总结，很值得看。

biser：入门的好书啊，不光菜鸟，老鸟也是很值得一读的，经典得很啊。

CSDN读书频道的本书读者评论

218.106.251.*："编程是门技术，更是一门艺术。"写得真好，看完了第一章，非常喜欢作者这种步步深入，循循善诱的写作方式。设计模式的书籍我也看过不少，不过以这么通俗的方式来讲解，特别是还有故事情节，幽默对白，的确给人耳目一新之感。让人忘记了是在学习设计模式，仿佛是在读小说一样，而读完之后，作者要讲的问题也就自然明白了。我猜测，这本书一定会引起不同凡响的轰动。

feng545218：这是我的专业老师强烈向我们推荐的一本适合我们的书，看后真的很好。既轻松幽默，又能记忆深刻，还形象化的理解了问题。

218.82.87.*：作者太TM有才了，怎么想到的，把23个模式变成23个MM，哈，恶搞了超级女生、潜规则、黄金甲，有意思。不过我最喜欢的还是第一章活字印刷那一段，哈，"喝酒唱歌，人生真爽。"面向对象经作者这么一说，的确通俗无比，比那种只会教条式的讲解面向对象的要强太多了。

abware：强啊，作者对OO和设计模式的理解有一定深度，不然不会写得这么深入浅出。

dangnilaoqu：不错呀。谁说理科就不能出文学家呀？

218.82.87.*：刚刚看完，正想发表一些感想，发现楼下的朋友说风格抄袭Head First，我不赞同这个观点。比如里面写到了三国曹操吟

诗，写到了近期国内的IT或娱乐事件。Head　First是好书，这一本没看完我还不知道是不是好书。但显然两本书风格是不一样的，我认为这是一本非常中国化的设计模式的技术书，如果国人都不支持国人作品，特别是已经不错的作品，那国人的技术又如何可以振兴？

Maciloveyou：写的很有特点，风格很像《水煮三国》，挺有意思的。

JunsGo：如果高校的教材也这么有趣那该多好啊！☹

58.246.92.*：用很简单的生活道理来告诉你深奥的编程思路。

作者博客的本书读者评论

肖雪平：昨晚有幸读到了你的《戏说面向对象编程》（也许就是上面提到的小菜编程成长记），非常欣赏。说实在话，一个人只是单纯的会写点文字，发发自己的牢骚；或者单纯的懂技术，开发能力强，我个人认为，算不得什么。难得的是，两者兼备。而你，做到了。能让外行一眼就爱上面向对象，爱上设计模式，并且一读就明白。何其难啊！这正是我一直努力的目标。因为我的职业就是软件开发老师。读过的晦涩、艰深、却百无一用的所谓教授专家所写的书，不敢说过千，但至少过百吧。看看你的，禁不住由衷的感慨：深入浅出的真谛，莫过于此！您的尝试非常有意义，希望您再接再厉，写出更多更好的文章，为我们国家的软件行业培养更多更实用的人才。

James　Liang：很有意思啊，之前就觉得你对这方面的东西有很深的研究和思维，现在居然还赋予激情幽默的故事，将它们说的很轻松和诙谐，不错的。所有的模式是为业务而服务的。期待这本书有更好的表现。嘿嘿……

横刀天笑：恭喜啊！很赞同你的有些观点：成为诗人后可能不需要刻意地按照某种模式去创作，但成为诗人前他们一定是认真地研究过成百上千的唐诗宋词、古今名句。我一直认为学习设计模式也要走从量变到质变这条路，不要把设计模式当圣经。

mekong：衷心恭喜您！真是应了是金子总会发光的！期待着！您能这样写书真是我等之福！您教的工厂模式是我最先学用的模式，逐渐领略了其优势而开始关注设计模式，所以陆续浏览过N本设计模式的书，总因晦涩难懂坚持不了50页、80页就弃之而沿用了笨拙但顺手的方法。现在终于看到由您写的这本包含这多设计模式的书，期待早早拜读……

newrain：这一系列文章真的不错，很不错的文章，非常喜欢这样闲侃式的把思想传授给我们。

肖卓耘：刚去dangdang了……说没货了……等一有货就买……（买

回去当小说看）说实话，写得太好了……能和《明朝那些事》相媲美……

小侯：很少有人能把设计模式介绍得清楚易懂，你做到了，而且很好，很试合我们中国人的口味，呵呵，可能说得有点大，不过很适合我，请继续努力，我会经常来光顾的。

棠棠**dotNet**：绝对经典！看到博客园有这样的文章，真是爽。长期关注楼主的文章。

scotoma：博主的文采和文章我一直都很喜欢，看到末尾博主所提到的每天的坚持，我会努力做到的，您的书我已经买了，像博主所说的，大学刚毕业很难找到工作，我感觉主要的原因是每个人的四年中的每一天的24小时是怎么花的，只要你努力的用好这些时间，我想四年会学到很多，还怕找不到工作吗？其实如果每个人都能够利用好时间，什么事情都可以做得相对完美的。期待博主的好文章。

ily：楼主写的东西很好，可能对有些大虾来说是小儿科，但我相信对大多数开发人员来说不仅是一种互相学习，也是一种如何学会更高效更合理开发的思路，继续支持你！

tom385：真的不错，一口气看完了，相当通俗易懂，学到很多东西！谢谢你，期待更多好文章！

Ruby113028：其实像这类的书还是很有必要的。很多技术方面的书都当成字典一样的查询，由于实在是太枯燥了，往往读了没多少时间就放在一旁休息了。一直在寻找有没有作者能把技术书写活了，不要像帮助文档那样的枯燥。唉，如果都像帮助文档那样，还不如看帮助呢。希望能多几本大话设计模式这样的书。没有必要一定要划归到技术类，在休闲的时候看看其实也蛮有趣的，还能在休闲的时候学到点东西。

张卫林：前段时间专程去书店买了，现在看了几章，非常不错。一本值得从头看到尾的书，也是一本能让人从头看到尾的书。

yarco：另外，再次感谢作者，实际上我能很明确的看懂UML图例的这个聚合现象，完全是因为您在本书里放的大雁和雁群的关系，我几乎在一霎那间就懂了UML：）

本书策划编辑对本书的评论

在出版这本书前，

我就知道这是一本好书，

我想象过它畅销的情景，

这，

是一回事。

在出版这本书后，

我真正看到这本书得到了读者的一致好评，

成为了一本畅销书，

这，

则完全是另外一回事。

给出几个在我写下上面这段文字时的数字吧：

当当网畅销排行榜第4名

卓越亚马逊畅销榜第5名

第二书店畅销榜第1名

CSDN 2007年度十大精品图书第1名

我是本书的策划编辑陈冰，给读者带去好书是我的责任。如果你有任何意见或建议想要告诉我，可以直接到我的博客（http://w3cbook.blog.sohu.com/）上留言。

今天是2008年1月31日，还有几天就要过大年了。在这里我代表我自己、本书作者程杰和清华大学出版社全体同仁给读者们拜年，希望每位读者的每一天都有过大年前快乐的心情。

前言

- ➢ 本书是一本程序集？NO。
- ➢ 本书是一本故事集？NO。
- ➢ 本书是一本通过故事讲述程序如何设计的方法集。
- ➢ 本书是给连Hello World都没写过的非程序员看的书吗？NO。
- ➢ 本书是给玩过穿孔纸带（0/1）、写过汇编、BASIC、C、C++、Delphi、Java、C#等语言，开发过覆盖全球、使用人数过亿、数百万行代码等大型系统的骨灰级程序员看的书吗？NO。
- ➢ 本书希望能给渴望了解OO世界的初学者、困惑于僵硬、脆弱、无法复用的代码编程体验者、一直打着OO编程的旗号，做着过程式开发的基于对象的编程实践者一些好的建议和提示。

本书起因

写本书源于我一次做培训的经历，学生大多是计算机专业的学生或有过一定经验的在职开发者。他们都知道类、方法、构造方法、甚至抽象类、接口等概念，并用Visual Studio写过不少的Windows或Web程序，可是当我提问为什么要用面向对象，它的好处在哪里时，却没有人能完整地讲得出来，多数人的反应是，概念知道的，就是表达不清楚。

针对于此，我就举了中国古代的四大发明中活字印刷的例子（见第1章），通过一个虚构的三国曹操做诗的情景，把面向对象的几大好处讲解了一下，学生普遍都感觉通俗易懂，觉得这样的教学比直接告诉面向对象有什么好处要更加容易理解和记忆。

这就使得我不断地思考这样一个问题，学一门技术是否需要趣味性、通俗性的引导。

我在思考中发现，看小说时，一般情况下我都可以完整地读完它，而阅读技术方面的图书，却很少有真正的每章每页的仔细阅读。尽管这两者是有很大区别，技术书中可能有不少知识是已经学会或暂时用不上的内容，但也不得不承认，小说之所以可以坚持读完是因为对它感兴趣，作者的文字吸引你。而有些技术书的枯燥乏味使得阅读产生了困难，通常读个前几章就留待以后再说了。

技术课的教学同样如此，除非学生是抱着极大的学习动机来参与其中，否则照本宣科的教学、枯燥乏味的讲解，学生一定会被庞杂的概念和复杂的逻辑搅晕了头脑，致使效果大打折扣。也正因为此，往往造成部分学生，学了四年的计算机编程，却可能连面向对象有什么好处都还说不清。

为什么不可以让技术书带点趣味性呢，哪怕这些趣味性与所讲的技术并不十分贴切，只要不是影响技术核心的本质，不产生重大的错误，让读者能轻松阅读它，并且有了一定的了解和感悟，这要比一本书写得高深无比，却被长期束之高阁要好得多。

也正是这个原因，本人开始了关于设计模式的趣味性写作的尝试。

本书读者

显然本书不是给无任何编程经验的人看的，对于想入这一行的朋友来说，找一门编程语言，从头开始或许才是正道。而本书也不太适合有了多年面向对象开发经验，对常用的设计模式了如指掌的人看的。毕竟这里更多的是一些基础性的东西。

我时常拿程序员的成长与足球运动员的成长做对比。

GoF的《设计模式》好比是世界顶级足球射门集锦，《重构》、《敏捷软件开发》、《设计模式解析》好比是一场场最精彩的足球比赛。我为之疯狂，为之着迷。可是我并不只是想做一个球迷（软件使用者），而是更希望自己能成为一个足球运动员（软件设计编程者），能够亲自上场比赛，并且最终能成为球星（软件架构师）。我仔细地阅读这些被誉为经典的著作，认真地实践其中代码，但是我总是半途而废、坚持不下去，我痛恨自己意志力的薄弱、憎恶自己无端地放弃，难道我真的就是那么的笨？

痛定思痛，反思悔过。我终于发现，贝利、马拉多纳不管老、胖是用来敬仰的，贝克汉姆、罗纳尔迪尼奥不管美、丑是用来欣赏的，但他们的球技……嗨，客气地说，是不容易学会的，客观地说，是不可能学得会的。为什么会这样？原来，我学习中缺了一个很重要的环节，我们在看到了精彩的球赛，欣赏球星高超球技的同时，却忽略了球星的成长过程。他们尽管有一定天分，但却也是从最底层通过努力一点一点慢慢显露出来的，我们需要的不仅仅是世界杯上的那定乾坤的一脚，更需要这一脚之前是如何练出那种神奇的方法，对于程序员来讲，精彩的代码是如何想出来的，要比看到精彩的代码更加令人期待。

本书显然不是培养足球明星（软件架构师）的俱乐部，而是训练足球基本功的学校，培训的是初学足球的小球员（面向对象的程序员），

本书希望的是读者阅读后可以打好面向对象的基础，从而更加容易并深入的去理解和感受GoF的《设计模式》以及其他大师作品的魅力。

本书定位

本书是在学习众多大师智慧结晶的图书作品、分享了网上多位朋友的实践经验的基础上，加之自己的编程感受写出来的。正如牛顿有句名言："如果说我比别人看得更远些，那是因为我站在了巨人的肩上。"但显然，本书并没有创造或发现什么模式，因此谈不上站在巨人肩膀上看得更远。所以作者更希望本书能成为一些准备攀登面向对象编程高峰的朋友的登山引路人、提携者，在您登山途中迷路时给予指引，在您峭壁攀岩摔跤时给予保护。

本书特色

本书有两个特色，第一特色是重视过程。看了太多的计算机编程类的图书，大多数书籍都是集中在讲授优秀的解决方案或者一个完美的程序样例，但对这些解决方案和程序的演变过程却重视不够，好书之所以好，就是因为作者可以站在学习者的角度去讲解问题所在，让学习门槛降低。《重构与模式》中有一句经典之语："如果想成为一名更优秀的软件设计师，了解优秀软件设计的演变过程比学习优秀设计本身更有价值，因为设计的演变过程中蕴藏着大智慧。"本人就希望能通过小菜与大鸟的对话，在不断地提问与回答过程中，在程序的不断重构演变中，把设计模式的学习门槛降低，让初学者可以更加容易地理解，为什么这样设计才是好，是如何想到这样设计的。

本书的第二个特色就是贴近生活。尽管编程是严谨的，不容大话和戏说。但生活却是多姿多彩的，而设计模式也不是完全孤立于现实世界而凭空想出来的理论。事实上所有的模式都可以在生活中找到对应。因此，通过主人公小菜和大鸟的对话，将求职、面试、工作、交友、投资、兼职、办公室文化、生活百味等等非常接近程序员生活原貌的场景写到了书中，用一个个小故事来引出模式，会让读者相对轻松地进入学习设计模式的状态。当然，此举的最大目的还是为了深入浅出，而非纯粹噱头。

本书内容

本书通篇都是以情景对话的形式，用一个又一个的小故事或编程示例来组织的。共分为四个部分。第一部分是面向对象的意义和好处以及

几个重要的设计原则，通过小菜面试的失败引出；第二部分是详细讲解23个设计模式；第三部分是对设计模式的总结，利用小菜梦到的超级模式大赛的场景，把所有的面向对象和模式概念都拟人化来趣味性的总结设计模式之间的异同和关键点。第四部分是附录，主要是针对对面向对象不熟悉读者的一个补充，通过一个例子的演变介绍了类、封装、继承、多态、接口、事件等概念。

本书人物及背景

小菜：原名蔡遥，22岁，上海人，上海某大学计算机专业大学四年级学生，成绩一般，考研刚结束，即将毕业，正求职找工作。

大鸟：原名李大辽，29岁，小菜的表哥，云南昆明人，毕业后长期从事软件开发和管理工作，近期到上海发展，借住小菜家在宝山的空套房内。小菜以向大鸟学习为由，也从市区父母家搬到宝山与大鸟同住。

本书研读方法

本书建议按顺序阅读，如果您感觉由于面向对象知识的匮乏，例如对继承、多态、接口、抽象类的理解不足，造成阅读上的困难，不妨先阅读附录一的"培训实习生——面向对象基础"部分，然后再从第1章开始阅读。如果您已经对不少设计模式熟悉，也不妨挑选不熟悉的模式章节阅读。

> 本书源代码下载地址：
> http://www.cnblogs.com/Files/cj723/BigTalkDesignPattenSourceCode.rar
> 也可到清华大学出版社网站（www.tup.com.cn）查找并下载本书源代码。

尽管本书中的代码都提供下载，但不经过读者的自己手动输入过程，其实阅读的效果是大打折扣的。强烈建议读者根据样例自己写程序，只有在运行出错，达不到预期效果时再查看本书提供的源程序，这样或许才是最好的学习方法。有问题可及时与我联系。我的电子邮箱是chengjielong@163.com，博客是http://cj723.cnblogs.com/。

本书中的很多精华都来自许多大师作品，建议读者通过笔记形式记录，这将有助于您的记忆和理解设计模式，增强最终的读书效果。

本书中出现的"[ ]"是表示句子摘自某书。例如，"策略模式（**Strategy**）：它定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化不会影响到使用算法的客户[DP]。"其

中“[DP]”表示此句摘自《设计模式：可复用面向对象软件的基础》，详细摘要说明请参看附录二。

本书中29章中的虚拟人物姓名都是软件编程中的专业术语，因此凡是专业术语被指向人物姓名的都用斜体字表示，以和实际术语区分。例如，“第一位是我们OOTV创始人，面向对象先生”，这里的斜体字面向对象指人名。

关于本书学习的疑问解答

➢ 看本书需要什么基础？

主要是C#或其他编程语言的基础知识，如变量、分支判断、循环、函数等编程基础，关于面向对象基础可参看本书的附录一。

➢ 设计模式是否有必要全部学一遍？

答案是，Yes！别被那些说什么设计模式大多用不上，根本不用全学的舆论所左右。尽管现在设计模式远远不止23种，对所有都有研究是不太容易的，但就像作者本人一样，在学习GoF总结的23个设计模式过程中，你会被那些编程大师们进行伟大的技术思想洗礼，不断增加自己对面向对象的深入理解，从而更好的把这种思想发扬光大。这就如同高中时学立体几何感觉没用，但当你装修好房子购买家俱时才知道，有空间感，懂得空间计算是如何的重要，你完全可能遇到买了一个大号的冰箱却放不进厨房，或买了开关门的衣橱（移门不占空间）却因床在旁边堵住了门而打不开的尴尬。

重要的不是你将来会不会用到这些模式，而是通过这些模式让你找到“封装变化”、“对象间松散耦合”、“针对接口编程”的感觉，从而设计出易维护、易扩展、易复用、灵活性好的程序。成为诗人后可能不需要刻意地按照某种模式去创作，但成为诗人前他们一定是认真地研究过成百上千的唐诗宋词、古今名句。

如果说，数学是思维的体操，那设计模式，就是面向对象编程思维的体操。

➢ 我学了设计模式后时常会过度设计，如何办？

作者建议，暂时现象，继续努力。

设计模式有四境界：

1．没学前是一点不懂，根本想不到用设计模式，设计的代码很糟糕；

2．学了几个模式后，很开心，于是到处想着要用自己学过的模式，于是时常造成误用模式而不自知；

3．学完全部模式时，感觉诸多模式极其相似，无法分清模式之间的差异，有困惑，但深知误用之害，应用之时有所犹豫；

4．灵活应用模式，甚至不应用具体的某种模式也能设计出非常优秀的代码，以达到无剑胜有剑的境界。

从作者本人的观点来说，不会用设计模式的人要远远超过过度使用设计模式的人，从这个角度讲，因为怕过度设计而不用设计模式显然是因噎废食。当你认识到自己有过度使用模式的时候，那就证明你已意识到问题的存在，只有通过不断的钻研和努力，你才能突破“不识庐山真面目，只缘身在此山中”的瓶颈，达到“会当凌绝顶，一览众山小”的境界。

编程语言的差异

本书讲的是面向对象设计模式，是用.NET中的C#语言编写，但本书并不是主要讲解C#语言或.NET框架的图书，因此本书同样适合Java、VB.NET、C＋＋等其他一些面向对象语言的读者阅读来学习设计模式。

就Java而言，主要差异来自C#对于子类继承父类或实现接口用的都是“：”，而Java中两者是有区别的。

当Cat继承抽象类Animal时，Java语法是

```
public class Cat extends Animal
```

当Superman实现接口IFly时，Java语法是

```
public class Superman implements IFly
```

然后Java中所有的方法都是虚拟的，因此不用指定new或是override修饰符。还有一些其他差异，但基本都不影响本书的阅读。

对于VB.NET的程序员，如果阅读困难，不妨去网上查找关于转换

C#与VB.Net语言的工具，比如http://www.kamalpatel.net/ConvertCSharp2VB.aspx，将下载本书的源代码转换后再进行阅读。

C＋＋的程序员，可能在语言上会有些差异，不过本书应该不会因为语言造成对面向对象思想的误读。

不是一个人在战斗

首先要感谢我的妻子李秀芳对我写作本书期间的全力支持，没有她的理解和鼓励，就不可能有本书的出版。而我们的宝宝也将在2008年初出生，希望等宝宝懂事后能知道，在宝宝的母亲怀胎过程中，宝宝的父亲也在为书的诞生而努力。也希望本书成为赠送给他或者她的最好的礼物。

父母的养育才有作者本人的今天，本书的出版，寻根溯源，也是父母用心教育的结果。养育之恩，没齿难忘。

本书起源于本人在“博客园”网站的博客http://cj723.cnblogs.com/中的一个连载文章《小菜编程成长记》。没想到连载引起了不小的反应，网友们普遍认为本人的这种技术写作方式新颖、有趣、喜欢看。正是因为众多网友的支持，本人有了要把GoF的23种设计模式全部成文的冲动。非常感谢这些在博客回复中鼓励我的朋友。

这里需要特别提及洪立人先生，他是本人在写书期间共同为理想奋斗的战友，写作也得到了他的大力支持和帮助，我写作的不少妙句也来自我们俩共同合作的网站http://www.miaoju.net。在此对他表示衷心的感谢。

写作过程中，本人参考了许多国内外大师的设计模式的著作。尤其是《设计模式》（作者：简称GoF的Erich Gamma，Richard Helm，Ralph Johnson，John Vlissides）、《设计模式解析》（作者：Alan Shalloway，James R. Trott）、《敏捷软件开发：原则、模式与实践》（作者：Robert C.Martin）、《重构——改善既有代码的设计》（作者：Martin Fowler）、《重构与模式》（作者：Joshua Kerievsky）、《Java与模式》（作者：阎宏）等等，没有他们的贡献，就没有本书的出版。也希望本书能成为更好阅读他们这些大师作品的前期 读物。

写作过程中，本人还参考了http://www.dofactory.com/ 关于23个设计模式的讲解，并引用了他们的结构图和基本代码。在博客园中的许多朋友，比如张逸、吕震宇、李会军、idior、Allen Lee的博文，MSDN SmartCast中李建忠的讲座，CSDN博客中的大卫、ai92的博文，网站J道www.jdon.com的版主banq的文章都给本人的写作提供了非常大的指引和帮助，在此表示感谢。另外博客园的双鱼座先生还对本人的部分代

码提出了整改意见，也表示衷心的谢意。详细参考资料与网站链接，见附录二。

事实上，由于本人长期有看书记读书笔记的习惯，所以书中引用笔记的内容，也极有可能是来自某本书或者某个朋友的博客、某个网站的文章。而本人已经无法一一说出其引用的地址，但这些作者的智慧同样对本书的写作带来了帮助，在此只能说声谢谢。

最后，对本书的责任编辑陈冰先生及清华大学出版社的相关工作人员，表示由衷的感谢。本书的出版离不开陈先生的指导和其他工作人员的辛勤工作。

程　杰
2007年7月

第1章　代码无错就是优？——简单工厂模式

1.1　面试受挫

小菜今年计算机专业大四了，学了不少软件开发方面的东西，也学着编了些小程序，踌躇满志，一心要找一个好单位。当投递了无数份简历后，终于收到了一个单位的面试通知，小菜欣喜若狂。

到了人家单位，前台小姐给了他一份题目，上面写着："请用C++、Java、C#或VB.NET任意一种面向对象语言实现一个计算器控制台程序，要求输入两个数和运算符号，得到结果。"

小菜一看，这个还不简单，三下五除二，10分钟不到，小菜写完了，感觉也没错误。交卷后，单位说一周内等通知吧。于是小菜只得耐心等待。可是半个月过去了，什么消息也没有，小菜很纳闷，我的代码实现了呀，为什么不给我机会呢。

时间：2月26日20点　　地点：大鸟房间　　人物：小菜、大鸟

小菜找到从事软件开发工作七年的表哥大鸟，请教原因，大鸟问了题目和了解了小菜代码的细节以后，哈哈大笑，说道："小菜呀小菜，你上当了，人家单位出题的意思，你完全都没明白，当然不会再联系你了。"

小菜说："我的代码有错吗？单位题目不就是要我实现一个计算器的代码吗，我这样写有什么问题。"

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("请输入数字A：");
        string A = Console.ReadLine();
        Console.Write("请选择运算符号(+、-、*、/)：");
```

```
            string B = Console.ReadLine();
            Console.Write("请输入数字B：");
            string C = Console.ReadLine();
            string D = "";
            if(B == "+")
                D = Convert.ToString(Convert.ToDouble(A)
            if(B == "-")
                D = Convert.ToString(Convert.ToDouble(A)
     Convert.ToDouble(C));
            if(B == "*")
                D = Convert.ToString(Convert.ToDouble(A)
            if(B == "/")
                D = Convert.ToString(Convert.ToDouble(A)
            Console.WriteLine("结果是：" + D);
        }
    }
```

1.2 初学者代码毛病

大鸟说："且先不说出题人的意思，单就你现在的代码，就有很多不足的地方需要改进。"

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("请输入数字A：");
        string A = Console.ReadLine();
        Console.Write("请选择运算符号(+、-、*、/)：");
        string B = Console.ReadLine();
        Console.Write("请输入数字B：");
        string C = Console.ReadLine();
        string D = "";

        if (B == "+")
            D = Convert.ToString(Convert.ToDouble(A) + Convert.ToDouble(C));
        if (B == "-")
            D = Convert.ToString(Convert.ToDouble(A) - Convert.ToDouble(C));
        if (B == "*")
            D = Convert.ToString(Convert.ToDouble(A) * Convert.ToDouble(C));
        if (B == "/")
            D = Convert.ToString(Convert.ToDouble(A) / Convert.ToDouble(C));
        Console.WriteLine("结果是：" + D);
    }
}
```

这样命名是非常不规范的

判断分支，你这样的写法，意味着每个条件都要做判断，等于计算机做了三次无用功

如果除数时，客户输入了0怎么办，如果用户输入的是字符符号而不是数字怎么办

1.3 代码规范

“哦，说得没错，这个我以前听老师说过，可是从来没有在意过，我马上改，改完再给你看看。”

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Console.Write("请输入数字A：");
            string strNumberA = Console.ReadLine();
            Console.Write("请选择运算符号(+、-、
*、/)：");
            string strOperate = Console.ReadLine();
            Console.Write("请输入数字B：");
            string strNumberB = Console.ReadLine();
            string strResult = "";
            switch(strOperate)
            {
                case "+":
                    strResult = Convert.ToString(Conv
                        + Convert.ToDouble(strNumberE
                    break;
                case "-":
                    strResult = Convert.ToString(Conv
                        -
 Convert.ToDouble(strNumberB));
                    break;
                case "*":
                    strResult = Convert.ToString(Conv
                        * Convert.ToDouble(strNumberE
                    break;
                case "/":
                    if(strNumberB != "0")
                        strResult = Convert.ToString(
                            / Convert.ToDouble(strN
                    else
                        strResult = "除数不能为0";
                    break;
```

```
                }
                Console.WriteLine（"结果
是：" + strResult）;
                Console.ReadLine（）;
            }
            catch（Exception ex）
            {
                Console.WriteLine（"您的输入有
错：" + ex.Message）;
            }
        }
    }
```

大鸟："吼吼，不错，不错，改得很快嘛？至少就目前代码来说，实现计算器是没有问题了，但这样写出的代码是否合出题人的意思呢？"

小菜："你的意思是面向对象？"

大鸟："哈，小菜非小菜也！"

1.4 面向对象编程

小菜："我明白了，他说用任意一种面向对象语言实现，那意思就是要用面向对象的编程方法去实现，对吗？OK，这个我学过，只不过当时我没想到而已。"

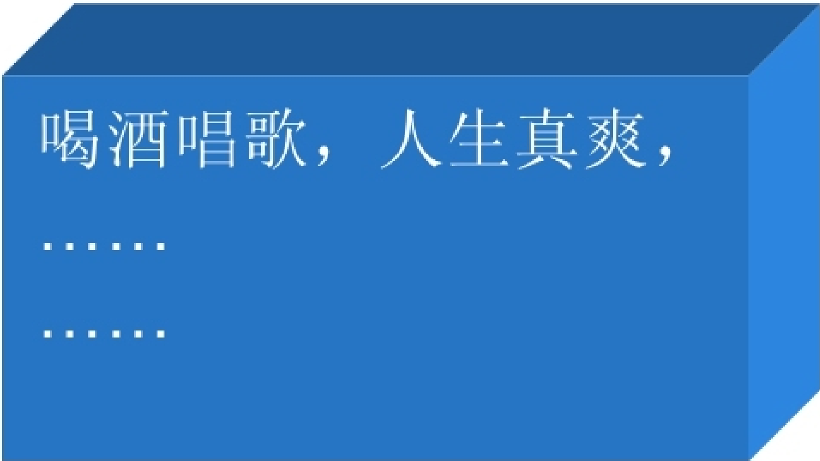
大鸟："所有编程初学者都会有这样的问题，就是碰到问题就直觉地用计算机能够理解的逻辑来描述和表达待解决的问题及具体的求解过程。这其实是用计算机的方式去思考，比如计算器这个程序，先要求输入两个数和运算符号，然后根据运算符号判断选择如何运算，得到结果，这本身没有错，但这样的思维却使得我们的程序只为满足实现当前的需求，程序不容易维护，不容易扩展，更不容易复用。从而达不到高质量代码的要求。"

小菜："鸟哥呀，我有点糊涂了，如何才能容易维护，容易扩展，又容易复用呢，能不能具体点？"

1.5 活字印刷，面向对象

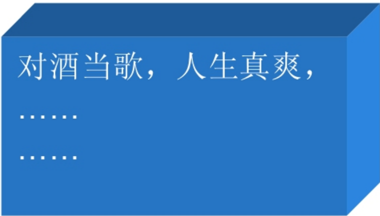
大鸟："这样吧，我给你讲个故事。你就明白了。"

"话说三国时期，曹操带领百万大军攻打东吴，大军在长江赤壁驻扎，军船连成一片，眼看就要灭掉东吴，统一天下，曹操大悦，于是大宴众文武，在酒席间，曹操诗性大发，不觉吟道：'喝酒唱歌，人生真爽。……'。众文武齐呼：'丞相好诗！'于是一臣子速命印刷工匠刻版印刷，以便流传天下。"



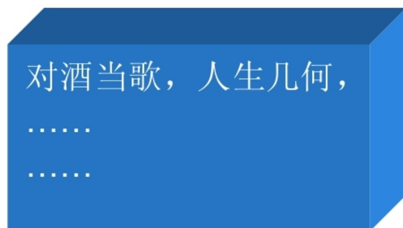
"样张出来给曹操一看，曹操感觉不妥，说道：'喝与唱，此话过俗，应改为'对酒当歌'较好！'，于是此臣就命工匠重新来过。工匠眼看连夜刻版之工，彻底白费，心中叫苦不迭。只得照办。"





"样张再次出来请曹操过目，曹操细细一品，觉得还是不好，说：'人生真爽太过直接，应改问语才够意境，因此应改为'对酒当歌，人生几何？……'当臣转告工匠之时，工匠晕倒……！"

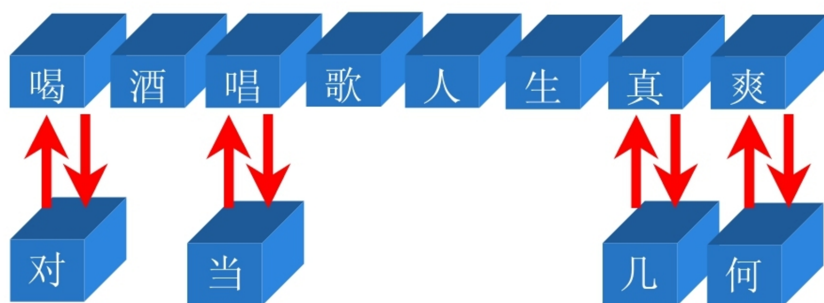




“小菜你说，这里面问题出在哪里？”大鸟问道。

小菜说：“是不是因为三国时期活字印刷还未发明，所以要改字的时候，就必须要整个刻板全部重新刻。”

大鸟：“说得好！如果是有了活字印刷，则只需更改四个字就可，其余工作都未白做。岂不妙哉。”



“第一，要改，只需更改要改之字，此为可维护；第二，这些字并非用完这次就无用，完全可以在后来的印刷中重复使用，此乃可复用；第三，此诗若要加字，只需另刻字加入即可，这是可扩展；第四，字的排列其实可能是竖排可能是横排，此时只需将活字移动就可做到满足排列需求，此是灵活性好。”

“而在活字印刷术出现之前，上面的四种特性都无法满足，要修改，必须重刻，要加字，必须重刻，要重新排列，必须重刻，印完这本书后，此版已无任何可再利用价值。”

小菜：“是的，小时候，我一直奇怪，为何火药、指南针、造纸术都是从无到有，从未知到发现的伟大发明，而活字印刷仅仅是从刻版印刷到活字印刷的一次技术上的进步，为何不是评印刷术为四大发明之一呢？原来活字印刷的成功是这个原因。”

1.6 面向对象的好处

大鸟："哈，这下你明白了？我以前也不懂，不过做了软件开发几年后，经历了太多的类似曹操这样的客户要改变需求，更改最初想法的事件，才逐渐明白当中的道理。其实客观地说，客户的要求也并不过份，不就是改几个字吗，但面对已完成的程序代码，却是需要几乎重头来过的尴尬，这实在是痛苦不堪。说白了，原因就是因为我们原先所写的程序，不容易维护，灵活性差，不容易扩展，更谈不上复用，因此面对需求变化，加班加点，对程序动大手术的那种无奈也就成了非常正常的事了。之后当我学习了面向对象的分析设计编程思想，开始考虑通过封装、继承、多态把程序的耦合度降低，传统印刷术的问题就在于所有的字都刻在同一版面上造成耦合度太高所致，开始用设计模式使得程序更加的灵活，容易修改，并且易于复用。体会到面向对象带来的好处，那种感觉应该就如同是一中国酒鬼第一次喝到了茅台，西洋酒鬼第一次喝到了XO一样，怎个爽字可形容呀。"

"是呀是呀，你说得没错，中国古代的四大发明，另三种应该都是科技的进步，伟大的创造或发现。而唯有活字印刷，实在是思想的成功，面向对象的胜利。"小菜也兴奋起来："你的意思是，面试公司出题的目的是要我写出容易维护，容易扩展，又容易复用的计算器程序？那该如何做呀？"

1.7 复制vs.复用

大鸟："比如说，我现在要求你再写一个Windows的计算器，你现在的代码能不能复用呢？"



小菜："那还不简单，把代码复制过去不就行了吗？改动又不大，不算麻烦。"

大鸟："小菜看来还是小菜呀，有人说初级程序员的工作就是Ctrl＋C和Ctrl＋V，这其实是非常不好的编码习惯，因为当你的代码中重复的代码多到一定程度，维护的时候，可能就是一场灾难。越大的系统，这种方式带来的问题越严重，编程有一原则，就是用尽可能的办法去避免重复。想想看，你写的这段代码，有哪些是和控制台无关的，而只是和计算器有关的？"

小菜："你的意思是分一个类出来？哦，对的，让计算和显示分开。"

1.8 业务的封装

大鸟："准确地说，就是让业务逻辑与界面逻辑分开，让它们之间的耦合度下降。只有分离开，才可以达到容易维护或扩展。"

小菜："让我来试试看。"

Operation运算类

```
public class Operation
{
    public static double GetResult(double numberA,
double numberB,string operate)
    {
        double result = 0d;
        switch(operate)
        {
            case "+":
                result = numberA + numberB;
                break;
            case "-":
                result = numberA - numberB;
                break;
            case "*":
                result = numberA * numberB;
                break;
            case "/":
                result = numberA / numberB;
                break;
        }
        return result;
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    try
```

```
        {
            Console.Write("请输入数字A：");
            string strNumberA = Console.ReadLine();
            Console.Write("请选择运算符号(+、-、
    *、/)：");
            string strOperate = Console.ReadLine();
            Console.Write("请输入数字B：");
            string strNumberB = Console.ReadLine();
            string strResult = "";
            strResult = Convert.ToString(Operation.GetRes
            Convert.ToDouble(strNumberB),
    strOperate));
            Console.WriteLine("结果
    是：" + strResult);
            Console.ReadLine();
        }
        catch(Exception ex)
        {
            Console.WriteLine("您的输入有
    错：" + ex.Message);
        }
    }
```

小菜："鸟哥，我写好了，你看看！"

大鸟："孺鸟可教也，写得不错，这样就完全把业务和界面分离了。"

小菜心中暗骂："你才是鸟呢。"口中说道："如果你现在要我写一个Windows应用程序的计算器，我就可以复用这个运算类（Operation）了。"

大鸟："不单是Windows程序，Web版程序需要运算可以用它，PDA、手机等需要移动系统的软件需要运算也可以用它呀。"

小菜："哈，面向对象不过如此。下回写类似代码不怕了。"

大鸟："别急，仅此而已，实在谈不上完全面向对象，你只用了面向对象三大特性中的一个，还有两个没用呢？"

小菜："面向对象三大特性不就是封装、继承和多态吗，这里我用到的应该是封装。这还不够吗？我实在看不出，这么小的程序如何用到继承。至于多态，其实我一直也不太了解它到底有什么好处，如何使用它。"

大鸟：“慢慢来，要学的东西多着呢，你好好想想该如何应用面向对象的继承和多态。”

1.9 紧耦合vs.松耦合

第二天。

小菜问道："你说计算器这样的小程序还可以用到面向对象三大特性？继承和多态怎么可能用得上，我实在不能理解。"

大鸟："小菜很有钻研精神嘛，好，今天我让你功力加深一级。你先要考虑一下，你昨天写的这个代码，能否做到很灵活的可修改和扩展呢？"

小菜："我已经把业务和界面分离了呀，这不是很灵活了吗？"

大鸟："那我问你，现在如果我希望增加一个开根（sqrt）运算，你如何改？"

小菜："那只需要改Operation类就行了，在switch中加一个分支就行了。"

大鸟："问题是你要加一个平方根运算，却需要让加减乘除的运算都得来参与编译，如果你一不小心，把加法运算改成了减法，这岂不是大大的糟糕。打个比方，如果现在公司要求你为公司的薪资管理系统做维护，原来只有技术人员（月薪），市场销售人员（底薪+提成），经理（年薪+股份）三种运算算法，现在要增加兼职工作人员（时薪）的算法，但按照你昨天的程序写法，公司就必须要把包含原三种算法的运算类给你，让你修改，你如果心中小算盘一打，'TMD，公司给我的工资这么低，我真是郁闷，这下有机会了'，于是你除了增加了兼职算法以外，在技术人员（月薪）算法中写了一句

```
if（员工是小菜）
{
    salary = salary * 1.1;
}
```

那就意味着，你的月薪每月都会增加10%（小心被抓去坐牢），本来是让你加一个功能，却使得原有的运行良好的功能代码产生了变化，这个风险太大了。你明白了吗？"

小菜："哦，你的意思是，我应该把加减乘除等运算分离，修改其中一个不影响另外的几个，增加运算算法也不影响其他代码，是这样吗？"

大鸟："自己想去吧，如何用继承和多态，你应该有感觉了。"

小菜："OK，我马上去写。"

Operation运算类

```
public class Operation
{
    private double _numberA = 0;
    private double _numberB = 0;

    public double NumberA
    {
        get  {  return _numberA;  }
        set  {  _numberA = value;  }
    }
    public double NumberB
    {
        get  {  return _numberB;  }
        set  {  _numberB = value;  }
    }
    public virtual double GetResult()
    {
        double result = 0;
        return result;
    }
}
```

加减乘除类

```
class OperationAdd : Operation
{
    public override double GetResult()
    {
        double result = 0;
        result = NumberA + NumberB;
        return result;
    }
}
```

加法类，继承运算类

```
class OperationSub : Operation
{
   public override double GetResult()
    {
        double result = 0;
        result = NumberA - NumberB;
        return result;
    }
}
```

减法类，继承运算类

```
class OperationMul : Operation
{
    public override double GetResult()
    {
        double result = 0;
        result = NumberA * NumberB;
        return result;
    }
}
```

乘法类，继承运算类

```
class OperationDiv : Operation
{
    public override double GetResult()
    {
        double result = 0;
        if (NumberB==0)
            throw new Exception("除数不能为0。");
        result = NumberA / NumberB;
        return result;
    }
}
```

除法类，继承运算类

小菜："大鸟哥，我按照你说的方法写出来了一部分，首先是一个运算类，它有两个Number属性，主要用于计算器的前后数，然后有一个

虚方法GetResult（ ），用于得到结果，然后我把加减乘除都写成了运算类的子类，继承它后，重写了GetResult（ ）方法，这样如果要修改任何一个算法，就不需要提供其他算法的代码了。但问题来了，我如何让计算器知道我是希望用哪一个算法呢？”

（作者注：以上代码读者如果感觉阅读吃力，说明您对继承、多态、虚方法、方法重写等概念的理解尚不够，建议先阅读本书的附录一，理解了这些基本概念后再继续往下阅读。）

1.10 简单工厂模式

大鸟："写得很不错嘛，大大超出我的想象了，你现在的问题其实就是如何去实例化对象的问题，哈，今天心情不错，再教你一招'简单工厂模式'，也就是说，到底要实例化谁，将来会不会增加实例化的对象，比如增加开根运算，这是很容易变化的地方，应该考虑用一个单独的类来做这个创造实例的过程，这就是工厂，来，我们看看这个类如何写。"

简单运算工厂类

```
public class OperationFactory
{
    public static Operation createOperate(string operate)
    {
        Operation oper = null;
        switch (operate)
        {
            case "+":
                oper = new OperationAdd();
                break;
            case "-":
                oper = new OperationSub();
                break;
            case "*":
                oper = new OperationMul();
                break;
            case "/":
                oper = new OperationDiv();
                break;
        }
        return oper;
    }
}
```

大鸟："哈，看到了吧，这样子，你只需要输入运算符号，工厂就实例化出合适的对象，通过多态，返回父类的方式实现了计算器的结果。"

客户端代码

```
Operation oper;
oper = OperationFactory.createOperate("+");
oper.NumberA = 1;
oper.NumberB = 2;
double result = oper.GetResult();
```

大鸟："哈，界面的实现就是这样的代码，不管你是控制台程序，Windows程序，Web程序，PDA或手机程序，都可以用这段代码来实现计算器的功能，如果有一天我们需要更改加法运算，我们只需要改哪里？"

小菜："改OperationAdd就可以了。"

大鸟："那么我们需要增加各种复杂运算，比如平方根，立方根，自然对数，正弦余弦等，如何做？"

小菜："只要增加相应的运算子类就可以了呀。"

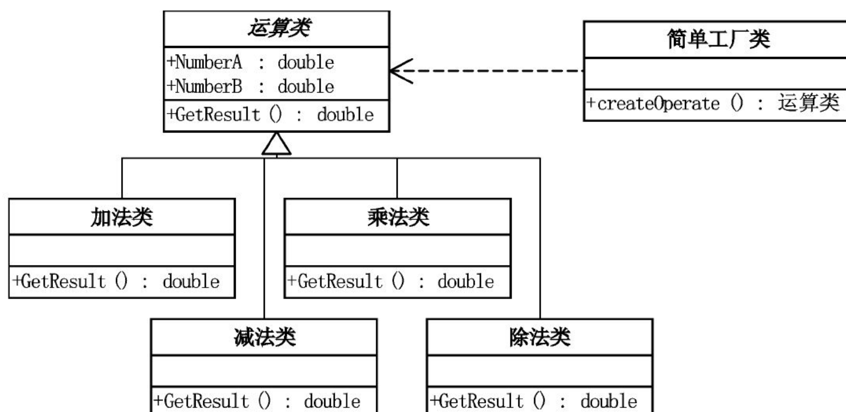
大鸟："嗯？够了吗？"

小菜："对了，还需要去修改运算类工厂，在switch中增加分支。"

大鸟："哈，那才对，那如果要修改界面呢？"

小菜："那就去改界面呀，关运算什么事呀。"

大鸟："好了，最后，我们来看看这几个类的结构图。"



1.11 UML类图

小菜："对了，我时常在一些技术书中看到这些类图表示，简单的还看得懂，有些标记我很容易混淆。要不你给我讲讲吧。"

大鸟："这个其实多看多用就熟悉了。我给你举一个例子，来看这样一幅图，其中就包括了UML类图中的基本图示法。"

UML类图图示样例

大鸟："首先你看那个'动物'矩形框，它就代表一个类（Class）。类图分三层，第一层显示类的名称，如果是抽象类，则就用斜体显示。第二层是类的特性，通常就是字段和属性。第三层是类的操作，通常是方法或行为。注意前面的符号，'+'表示public，'-'表示private，'#'表示protected。"

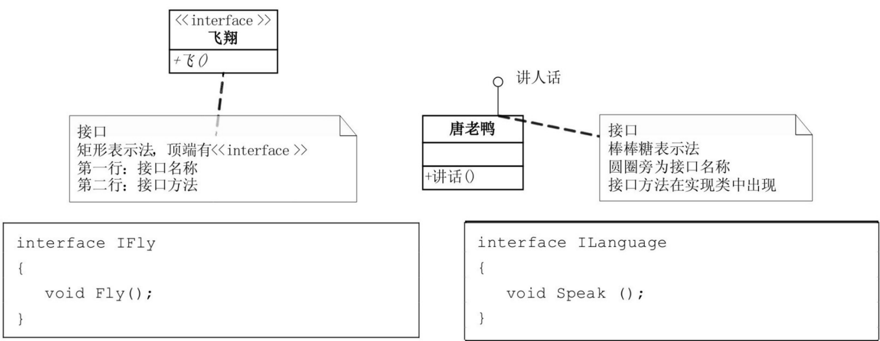
动物
+有生命
+新陈代谢(in o2 ： 氧气, in water ： 水) +繁殖()

大鸟：“然后注意左下角的‘飞翔’，它表示一个接口图，与类图的区别主要是顶端有＜＜interface＞＞显示。第一行是接口名称，第二行是接口方法。接口还有另一种表示方法，俗称棒棒糖表示法，比如图中的唐老鸭类就是实现了‘讲人话’的接口。”

小菜：“为什么要是‘讲人话’？”

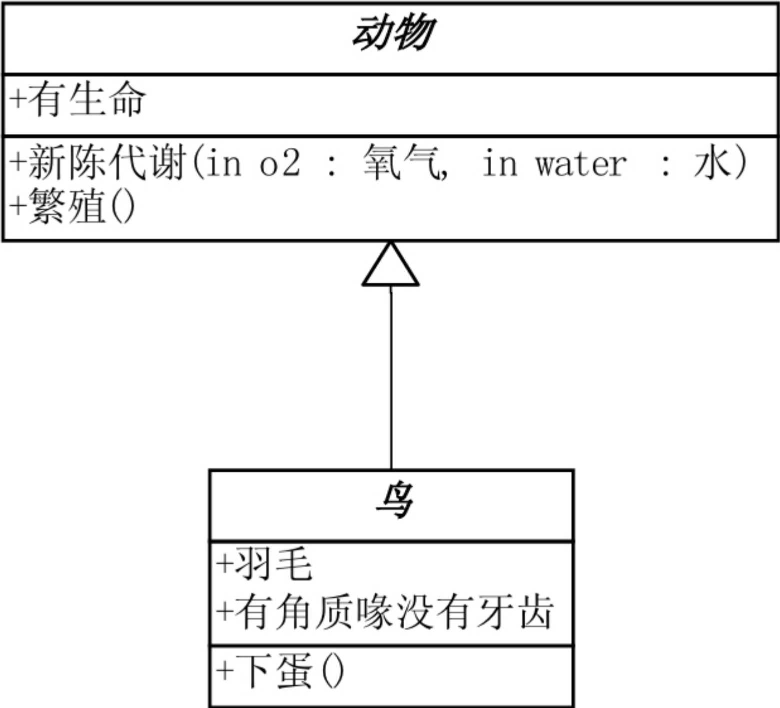
大鸟：“鸭子本来也有语言，只不过只有唐老鸭是能讲人话的鸭子。”

小菜：“有道理。”

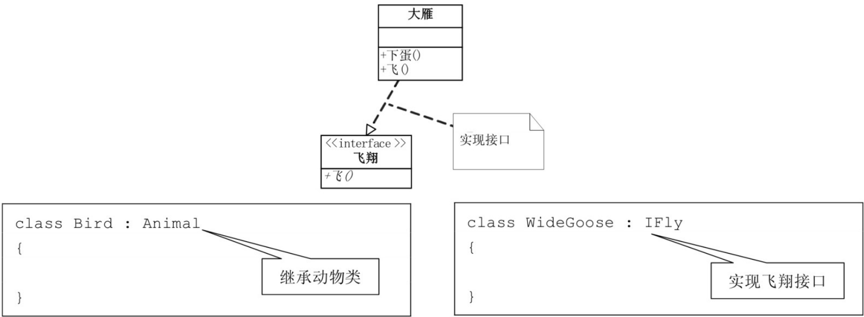


大鸟：“接下来就可讲类与类，类与接口之间的关系了。你可首先注意动物、鸟、鸭、唐老鸭之间关系符号。”

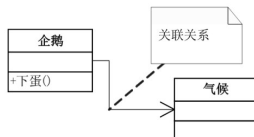
小菜：“明白了，它们都是继承的关系，继承关系用空心三角形＋实线来表示。”



大鸟："我举的几种鸟中，大雁是最能飞的，我让它实现了飞翔接口。实现接口用空心三角形＋虚线来表示。"



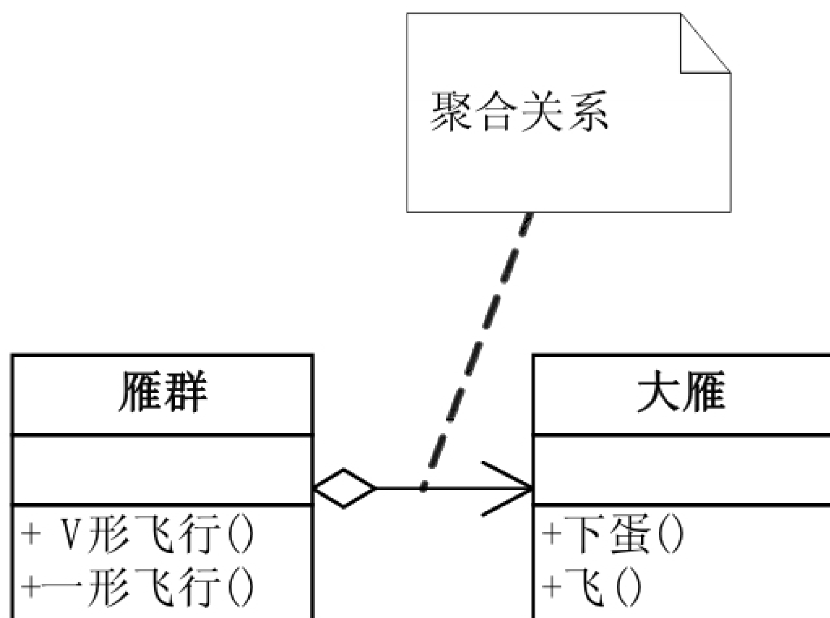
大鸟："你看企鹅和气候两个类，企鹅是很特别的鸟，会游不会飞。更重要的是，它与气候有很大的关联。我们不去讨论为什么北极没有企鹅，为什么它们要每年长途跋涉。总之，企鹅需要'知道'气候的变化，需要'了解'气候规律。当一个类'知道'另一个类时，可以用关联（association）。关联关系用实线箭头来表示。"




```
class Penguin : Bird
{
   private Climate climate;
}
```

在企鹅 Penguin 中，引用到气候 Climate 对象

大鸟："我们再来看大雁与雁群这两个类，大雁是群居动物，每只大雁都是属于一个雁群，一个雁群可以有多只大雁。所以它们之间就满足聚合（Aggregation）关系。聚合表示一种弱的'拥有'关系，体现的是**A**对象可以包含**B**对象，但**B**对象不是**A**对象的一部分[DPE]（DPE表示此句摘自《设计模式》（第2版），详细摘要说明见附录二）。聚合关系用空心的菱形＋实线箭头来表示。"

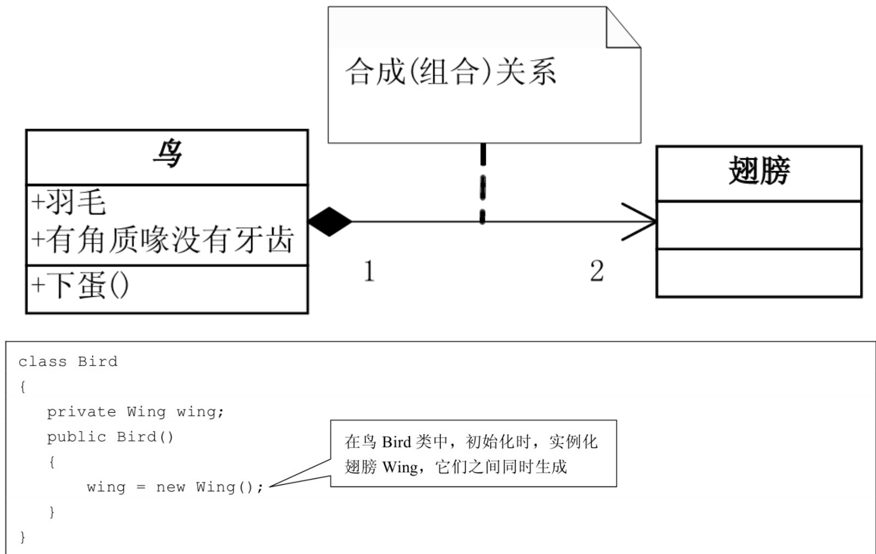



```
class WideGooseAggregate
{
   private WideGoose[] arrayWideGoose;
}
```

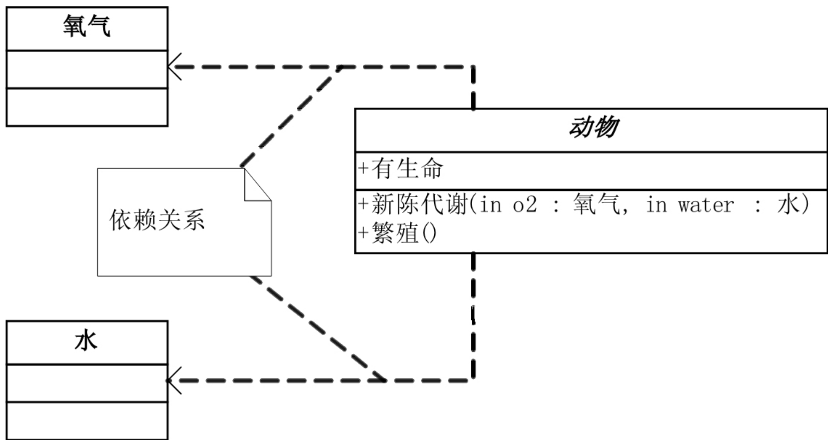
在雁群 WideGooseAggregate 类中，有大雁数组对象 arrayWideGoose

大鸟："合成（**Composition**，也有翻译成'组合'的）是一种强的'拥有'关系，体现了严格的部分和整体的关系，部分和整体的生命周期一样[DPE]。在这里鸟和其翅膀就是合成（组合）关系，因为它们是部分和整体的关系，并且翅膀和鸟的生命周期是相同的。合成关系用实心的菱形＋实线箭头来表示。另外，你会注意到合成关系的连线两端还有一个

数字‘1’和数字‘2’，这被称为基数。表明这一端的类可以有几个实例，很显然，一个鸟应该有两只翅膀。如果一个类可能有无数个实例，则就用‘n’来表示。关联关系、聚合关系也可以有基数的。”



大鸟：“动物几大特征，比如有新陈代谢，能繁殖。而动物要有生命力，需要氧气、水以及食物等。也就是说，动物依赖于氧气和水。他们之间是依赖关系（Dependency），用虚线箭头来表示。”




```
abstract class Animal
{
        public Metabolism(Oxygen oxygen,
```

```
    Water water)
            {

            }
    }
```

小菜："啊，看来UML类图也不算难呀。回想那天我面试题写的代码，我终于明白我为什么写得不成功了，原来一个小小的计算器也可以写出这么精彩的代码，谢谢大鸟。"

大鸟："吼吼，记住哦，编程是一门技术，更加是一门艺术，不能只满足于写完代码运行结果正确就完事，时常考虑如何让代码更加简练，更加容易维护，容易扩展和复用，只有这样才可以真正得到提高。写出优雅的代码真的是一种很爽的事情。UML类图也不是一学就会的，需要有一个慢慢熟练的过程。所谓学无止境，其实这才是理解面向对象的开始呢。"

第2章　商场促销——策略模式

2.1　商场收银软件

时间：2月27日22点　　地点：大鸟房间　　人物：小菜、大鸟

“小菜，给你出个作业，做一个商场收银软件，营业员根据客户所购买商品的单价和数量，向客户收费。”

“就这个？没问题呀。”小菜说，“用两个文本框来输入单价和数量，一个确定按钮来算出每种商品的费用，用个列表框来记录商品的清单，一个标签来记录总计，对，还需要一个重置按钮来重新开始，不就行了？！”

商场收银系统v1.0关键代码如下：

```
double total = 0.0d;

private void btnOk_Click(object sender, EventArgs e)
{

    double totalPrices=Convert.ToDouble(txtPrice.Text) * Convert.ToDouble(txtNum.Text);

    total = total + totalPrices;

    lbxList.Items.Add("单价：" +txtPrice.Text+" 数量：" 
    +txtNum.Text+" 合计：" +totalPrices.ToString());


    lblResult.Text = total.ToString();
}
```

声明一个 double 变量 total 来计算总计

声明一个 double 变量 totalPrices 来计算每个商品的单价（txtPrice）* 数量(txtNum)后的合计

将每个商品合计计入总计

在列表框中显示信息

在 lblResult 标签上显示总计数

“大鸟，”小菜叫道，“来看看，这不就是你要的收银软件吗？我不到半小时就搞定了。”

“哈哈，很快嘛，”大鸟说着，看了看小菜的代码。接着说：“现在我要求商场对商品搞活动，所有的商品打八折。”

“那不就是在totalPrices后面乘以一个0.8吗？”

“小子，难道商场活动结束，不打折了，你还要再改一遍程序代码，然后再用改后的程序去把所有机器全部安装一次吗？再说，还有可能因为周年庆，打五折的情况，你怎么办？”

小菜不好意思道：“啊，我想得是简单了点。其实只要加一个下拉选择框就可以解决你说的问题。”

大鸟微笑不语。

2.2 增加打折

商场收银系统v1.1关键代码如下：

```
double total = 0.0d;

private void Form1_Load(object sender, EventArgs e)
{
    cbxType.Items.AddRange(new object[] {"正常收费","打八折","打七折","打五折"});
    cbxType.SelectedIndex = 0;
}                                                    在ComboBox中
                                                     加下拉选项

private void btnOk_Click(object sender, EventArgs e)
{
    double totalPrices=0d;
    switch(cbxType.SelectedIndex)                 根据选项决定
    {                                             打折额度
        case 0:
            totalPrices=Convert.ToDouble(txtPrice.Text)*Convert.ToDouble(txtNum.Text);
            break;
        case 1:
            totalPrices=Convert.ToDouble(txtPrice.Text)*Convert.ToDouble(txtNum.Text)*0.8;
            break;
        case 2:
            totalPrices=Convert.ToDouble(txtPrice.Text)*Convert.ToDouble(txtNum.Text)*0.7;
            break;
        case 3:
            totalPrices=Convert.ToDouble(txtPrice.Text)*Convert.ToDouble(txtNum.Text)*0.5;
            break;
    }
    total = total + totalPrices;
    lbxList.Items.Add("单价：" + txtPrice.Text + " 数量：" + txtNum.Text
        + " "+cbxType.SelectedItem+ " 合计：" + totalPrices.ToString());
    lblResult.Text = total.ToString();
}
```

“这下可以了吧，只要我事先把商场可能的打折都做成下拉选择框的项，要变化的可能性就小多了。”小菜说道。

商场收银系统

单价：1000　　确定

数量：1　　重置

计算方式：打8折

单价：1000 数量：1 正常收费 合计：1000
单价：1000 数量：1 满300返100 合计：700
单价：1000 数量：1 打8折 合计：800

总计：2500

“这比刚才灵活性上是好多了，不过重复代码很多，像Convert.ToDouble（），你这里就写了8遍，而且4个分支要执行的语句除了打折多少以外几乎没什么不同，应该考虑重构一下。不过这还不是最主要的，现在我的需求又来了，商场的活动加大，需要有满300返100的促销算法，你说怎么办？”

“满300返100，那要是700就要返200了？这个必须要写函数了吧？”

“小菜呀，看来之前教你的白教了，这里面看不出什么名堂吗？”

“哦！我想起来了，你的意思是简单工厂模式是吧，对的对的，我可以先写一个父类，再继承它实现多个打折和返利的子类，利用多态，完成这个代码。”

“你打算写几个子类？”

“根据需求呀，比如八折、七折、五折、满300送100、满200送

50……要几个写几个。”

“小菜又不动脑子了，有必要这样吗？如果我现在要三折，我要满300送80，你难道再去加子类？你不想想看，这当中哪些是相同的，哪些是不同的？”

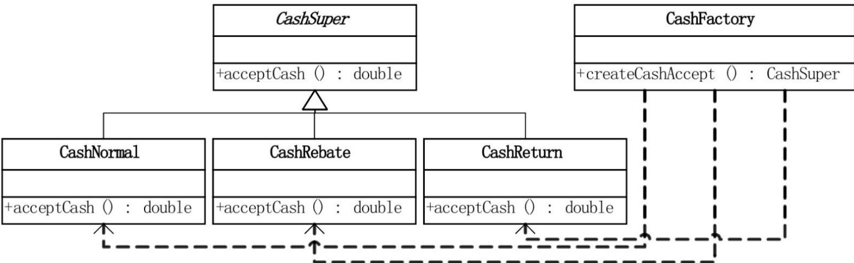
2.3 简单工厂实现

“对的，这里打折基本都是一样的，只要有个初始化参数就可以了。满几送几的，需要两个参数才行，明白，现在看来不麻烦了。”

“面向对象的编程，并不是类越多越好，类的划分是为了封装，但分类的基础是抽象，具有相同属性和功能的对象的抽象集合才是类。打一折和打九折只是形式的不同，抽象分析出来，所有的打折算法都是一样的，所以打折算法应该是一个类。好了，空话已说了太多，写出来才是真的懂。”

大约1个小时后，小菜交出了第三份的作业。

代码结构图



现金收费抽象类

```
abstract class CashSuper
{
    public abstract double acceptCash(double money);
}
```

现金收取超类的抽象方法，收取现金，参数为原价，返回为当前价

正常收费子类

```
class CashNormal : CashSuper
{
    public override double acceptCash(double money)
    {
        return money;
    }
}
```

正常收费，原价返回

打折收费子类

```
class CashRebate : CashSuper
{
    private double moneyRebate = 1d;
    public CashRebate(string moneyRebate)
    {
        this.moneyRebate = double.Parse(moneyRebate);
    }

    public override double acceptCash(double money)
    {
        return money * moneyRebate;
    }
}
```

打折收费，初始化时，必需要输入折扣率，如八折，就是 0.8

返利收费子类

```
class CashReturn : CashSuper
{
    private double moneyCondition = 0.0d;
    private double moneyReturn = 0.0d;
    public CashReturn(string moneyCondition, string moneyReturn)
    {
        this.moneyCondition = double.Parse(moneyCondition);
        this.moneyReturn = double.Parse(moneyReturn);
    }
    public override double acceptCash(double money)
    {
        double result = money;
        if (money >= moneyCondition)
            result = money - Math.Floor(money / moneyCondition) * moneyReturn;

        return result;
    }
}
```

返利收费，初始化时必须要输入返利条件和返利值，比如满 300 返 100，则 moneyCondition 为 300，moneyReturn 为 100

若大于返利条件，则需要减去返利值

现金收费工厂类

```
class CashFactory
{
    public static CashSuper createCashAccept(string type)   现金收取工厂
    {
        CashSuper cs = null;
        switch (type)
        {
            case "正常收费":
                cs = new CashNormal();          根据条件返回相应的对象
                break;
            case "满 300 返 100":
                CashReturn cr1 = new CashReturn("300", "100");
                cs = cr1;
                break;
            case "打 8 折":
                CashRebate cr2 = new CashRebate("0.8");
                cs = cr2;
                break;

          }
          return cs;
      }
  }
```

客户端程序主要部分

```
//客户端窗体程序（主要部分）
double total = 0.0d;
private void btnOk_Click(object sender, EventArgs e)        利用简单工厂模式根据下拉选择框，生成相应的对象
{
    CashSuper csuper = CashFactory.createCashAccept(cbxType.SelectedItem.ToString());
    double totalPrices = 0d;
    totalPrices = csuper.acceptCash(Convert.ToDouble(txtPrice.Text)
    * Convert.ToDouble(txtNum.Text));
    total = total + totalPrices;
    lbxList.Items.Add("单价: " + txtPrice.Text + " 数量: " + txtNum.Text + " "
        + cbxType.SelectedItem + " 合计: " + totalPrices.ToString());   通过多态，可以得到收取费用的结果
    lblResult.Text = total.ToString();
}
```

“大鸟，搞定，这次无论你要怎么改，我都可以简单处理就行了。”小菜自信满满地说。

“是吗，我要是需要打五折和满500送200的促销活动，如何办？”

“只要在现金工厂当中加两个条件，在界面的下拉选项框里加两项，就OK了。”

“现金工厂？！你当是生产钞票呀。是收费对象生成工厂才准确。说得不错，如果我现在需要增加一种商场促销手段，满100积分10点，以后积分到一定时候可以领取奖品如何做？”

“有了工厂，何难？加一个积分算法，构造方法有两个参数：条件和返点，让它继承CashSuper，再到现金工厂，哦，不对，是收费对象生成工厂里增加满100积分10点的分支条件，再到界面稍加改动，就行

了。”

“嗯，不错。你对简单工厂用得很熟练了嘛。”大鸟接着说：“简单工厂模式虽然也能解决这个问题，但这个模式只是解决对象的创建问题，而且由于工厂本身包括了所有的收费方式，商场是可能经常性地更改打折额度和返利额度，每次维护或扩展收费方式都要改动这个工厂，以致代码需重新编译部署，这真的是很糟糕的处理方式，所以用它不是最好的办法。面对算法的时常变动，应该有更好的办法。好好去研究一下其他的设计模式，你会找到答案的。”

小菜进入了沉思中……

2.4　策略模式

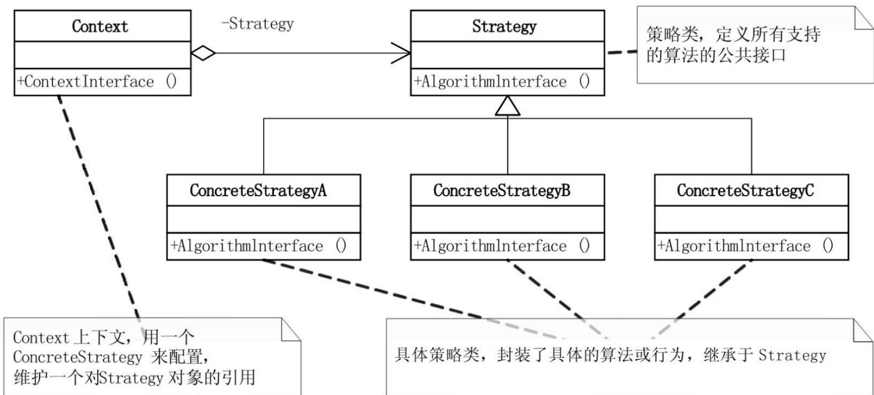
时间：2月28日19点　　地点：大鸟房间　　人物：小菜、大鸟

小菜次日来找大鸟，说："我找到相关的设计模式了，应该是策略模式（Strategy）。策略模式定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化，不会影响到使用算法的客户。看来商场收银系统应该考虑用策略模式？"

> **策略模式（Strategy）**：它定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化，不会影响到使用算法的客户。[DP]

"你问我？你说呢？"大鸟笑道，"商场收银时如何促销，用打折还是返利，其实都是一些算法，用工厂来生成算法对象，这没有错，但算法本身只是一种策略，最重要的是这些算法是随时都可能互相替换的，这就是变化点，而封装变化点是我们面向对象的一种很重要的思维方式。我们来看看策略模式的结构图和基本代码。"

策略模式（**Strategy**）结构图



Strategy类，定义所有支持的算法的公共接口

```
//抽象算法类
abstract class Strategy
{
```

```
    //算法方法
    public abstract void AlgorithmInterface();
}
```

ConcreteStrategy，封装了具体的算法或行为，继承于Strategy

```
//具体算法A
class ConcreteStrategyA : Strategy
{
    //算法A实现方法
    public override void AlgorithmInterface()
    {
        Console.WriteLine("算法A实现");
    }
}
//具体算法B
class ConcreteStrategyB : Strategy
{
    //算法B实现方法
    public override void AlgorithmInterface()
    {
        Console.WriteLine("算法B实现");
    }
}
//具体算法C
class ConcreteStrategyC : Strategy
{
    //算法C实现方法
    public override void AlgorithmInterface()
    {
        Console.WriteLine("算法C实现");
    }
}
```

Context，用一个ConcreteStrategy来配置，维护一个对Strategy对象的引用。

```
//上下文
class Context
{
    Strategy strategy;
    public Context(Strategy strategy)
    {
        this.strategy = strategy;
    }
    //上下文接口
    public void ContextInterface()
    {
        strategy.AlgorithmInterface();
    }
}
```

初始化时，传入具体的策略对象

根据具体的策略对象，调用其算法的方法

客户端代码

```
static void Main(string[] args)
{
    Context context;

    context = new Context(new ConcreteStrategyA());
    context.ContextInterface();

    context = new Context(new ConcreteStrategyB());
    context.ContextInterface();

    context = new Context(new ConcreteStrategyC());
    context.ContextInterface();

    Console.Read();
}
```

由于实例化不同的策略，所以最终在调用 context.ContextInterface();时，所获得的结果就不尽相同

2.5 策略模式实现

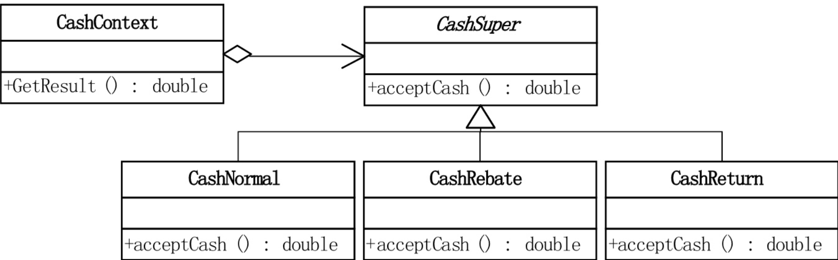
“我明白了，”小菜说，“我昨天写的CashSuper就是抽象策略，而正常收费CashNormal、打折收费CashRebate和返利收费CashReturn就是三个具体策略，也就是策略模式中说的具体算法，对吧？”

“是的，来吧，你模仿策略模式的基本代码，改写一下你的程序。”

“其实不麻烦，原来写的CashSuper、CashNormal、CashRebate和CashReturn都不用更改了，只要加一个CashContext类，并改写一下客户端就行了。”

商场收银系统v1.2

代码结构图



CashContext类

```
class CashContext
{
    private CashSuper cs;

    public CashContext(CashSuper csuper)
    {
        this.cs = csuper;
    }

    public double GetResult(double money)
    {
        return cs.acceptCash(money);
    }
}
```

声明一个 CashSuper 对象

通过构造方法，传入具体的收费策略

根据收费策略的不同，获得计算结果

客户端主要代码

```
double total = 0.0d;//用于总计
private void btnOk_Click(object sender, EventArgs e)
{
    CashContext cc=null;
    switch (cbxType.SelectedItem.ToString())
    {
        case "正常收费":
            cc = new CashContext(new CashNormal());
            break;
        case "满 300 返 100":
            cc = new CashContext(new CashReturn("300", "100"));
            break;
        case "打 8 折":
            cc = new CashContext(new CashRebate("0.8"));
            break;
    }

    double totalPrices = 0d;
    totalPrices = cc.GetResult(Convert.ToDouble(txtPrice.Text) * Convert.ToDouble(txtNum.Text));
    total = total + totalPrices;
    lbxList.Items.Add("单价: " + txtPrice.Text + " 数量: " + txtNum.Text
                + " "+cbxType.SelectedItem+ " 合计: " + totalPrices.ToString());
    lblResult.Text = total.ToString();
}
```

根据下拉选择框，将相应的策略对象作为参数传入 CashContext 的对象中

通过对 Context 的 GetResult 方法的调用，可以得到收取费用的结果，让具体算法与客户进行了隔离

“大鸟，代码是模仿着写出来了。但我感觉这样子做不又回到了原来的老路了吗，在客户端去判断用哪一个算法？”

“是的，但是你有没有什么好办法，把这个判断的过程从客户端程序转移走呢？”

“转移？不明白，原来我用简单工厂是可以转移的，现在这样子如何做到？”

“难道简单工厂就一定要是一个单独的类吗？难道不可以与策略模式的Context结合？”

“哦，我明白你的意思了。我试试看。”

2.6 策略与简单工厂结合

改造后的CashContext

```
class CashContext
{
    CashSuper cs = null;

    public CashContext(string type)
    {
        switch (type)
        {
            case "正常收费":
                CashNormal cs0 = new CashNormal();
                cs=cs0
                break;
            case "满 300 返 100":
                CashReturn cr1 = new CashReturn("300", "100");
                cs = cr1;
                break;
            case "打 8 折":
                CashRebate cr2 = new CashRebate("0.8");
                cs = cr2;
                break;
        }
    }

    public double GetResult(double money)
    {
        return cs.acceptCash(money);
    }
}
```


客户端代码

```
//客户端窗体程序（主要部分）
double total = 0.0d;
private void btnOk_Click(object sender, EventArgs e)
{
    CashContext csuper =new CashContext(cbxType.SelectedItem.ToString());
    double totalPrices = 0d;
    totalPrices =
        csuper.GetResult(Convert.ToDouble(txtPrice.Text) * Convert.ToDouble(txtNum.Text));
    total = total + totalPrices;
    lbxList.Items.Add("单价: " + txtPrice.Text + " 数量: " + txtNum.Text + " "
               + cbxType.SelectedItem + " 合计: " + totalPrices.ToString());
    lblResult.Text = total.ToString();
}
```


“嗯，原来简单工厂模式并非只有建一个工厂类的做法，还可以这样子做。此时比刚才的模仿策略模式的写法要清楚多了，客户端代码简单明了。”

“那和你写的简单工厂的客户端代码比呢？观察一下，找出它们的不同之处。”

```
//简单工厂模式的用法
CashSuper csuper = CashFactory.createCashAccept(cbxTy

...=csuper.GetResult(...)
```

```
//策略模式与简单工厂结合的用法
CashContext csuper =new CashContext(cbxType.SelectedI

...=csuper.GetResult(...);
```

“你的意思是说，简单工厂模式我需要让客户端认识两个类，CashSuper和CashFactory，而策略模式与简单工厂结合的用法，客户端就只需要认识一个类CashContext就可以了。耦合更加降低。”

“说得没错，我们在客户端实例化的是CashContext的对象，调用的是CashContext的方法GetResult，这使得具体的收费算法彻底地与客户端分离。连算法的父类CashSuper都不让客户端认识了。”

2.7 策略模式解析

“回过头来反思一下策略模式，策略模式是一种定义一系列算法的方法，从概念上来看，所有这些算法完成的都是相同的工作，只是实现不同，它可以以相同的方式调用所有的算法，减少了各种算法类与使用算法类之间的耦合 **[DPE]** 。”大鸟总结道。

“策略模式还有些什么优点？”小菜问道。

“策略模式的**Strategy**类层次为**Context**定义了一系列的可供重用的算法或行为。继承有助于析取出这些算法中的公共功能 **[DP]** 。对于打折、返利或者其他的算法，其实都是对实际商品收费的一种计算方式，通过继承，可以得到它们的公共功能，你说这公共功能指什么？”

“公共的功能就是获得计算费用的结果GetResult，这使得算法间有了抽象的父类CashSuper。”

“对，很好。另外一个策略模式的优点是简化了单元测试，因为每个算法都有自己的类，可以通过自己的接口单独测试 **[DPE]** 。”

“每个算法可保证它没有错误，修改其中任一个时也不会影响其他的算法。这真的是非常好。”

“哈，小菜今天表现不错，我所想的你都想到了。”大鸟表扬了小菜，“还有，在最开始编程时，你不得不在客户端的代码中为了判断用哪一个算法计算而用了switch条件分支，这也是正常的。因为，当不同的行为堆砌在一个类中时，就很难避免使用条件语句来选择合适的行为。将这些行为封装在一个个独立的**Strategy**类中，可以在使用这些行为的类中消除条件语句 **[DP]** 。就商场收银系统的例子而言，在客户端的代码中就消除条件语句，避免了大量的判断。这是非常重要的进展。你能用一句话来概况这个优点吗？”大鸟总结后问道。

“策略模式封装了变化。”小菜快速而坚定的说。

“说得非常好，策略模式就是用来封装算法的，但在实践中，我们发现可以用它来封装几乎任何类型的规则，只要在分析过程中听到需要在不同时间应用不同的业务规则，就可以考虑使用策略模式处理这种变化的可能性 **[DPE]** ”。

“但我感觉在基本的策略模式中，选择所用具体实现的职责由客户端对象承担，并转给策略模式的**Context**对象 **[DPE]** 。这本身并没有解除客户端需要选择判断的压力，而策略模式与简单工厂模式结合后，选择具体实现的职责也可以由Context来承担，这就最大化地减轻了客户端的职责。”

“是的，这已经比起初的策略模式好用了，不过，它依然不够完美。”

“哦，还有什么不足吗？”

“因为在CashContext里还是用到了switch，也就是说，如果我们需要增加一种算法，比如‘满200送50’，你就必须要更改CashContext中的switch代码，这总还是让人很不爽呀。”

“那你说怎么办，有需求就得改呀，任何需求的变更都是需要成本的。”

“但是成本的高低还是有差异的。高手和菜鸟的区别就是高手可以花同样的代价获得最大的收益或者说做同样的事花最小的代价。面对同样的需求，当然是改动越小越好。”

“你的意思是说，还有更好的办法？”

“当然。这个办法就是用到了反射技术，不是常有人讲，‘反射反射，程序员的快乐’，不过今天就不讲了，以后会再提它的。”

“反射真有这么神奇？”小菜疑惑地望向了远方。

（注：在抽象工厂模式章节有对反射的讲解）

第3章　拍摄UFO——单一职责原则

3.1　新手机

时间：2月28日18点　　地点：小菜大鸟居住的小区附近　　人物：小菜、大鸟

大鸟小菜晚上晚饭过后，在外面散步。

大鸟：“小菜，刚换的手机感觉如何？”

小菜：“哈，当然是怎个爽字了得，可以听音乐、玩游戏、拍照、摄像，功能全着呢！”

大鸟：“你们这些小年轻，只会赶时髦，手机要那么多功能干吗？能打电话就可以了。”

小菜：“这你就不懂了吧，比如你出门旅游，数码相机一定要的吧，拍照是最起码的旅游需求；有摄像机会更好，动的影像不是更有保留价值吗；一路上无聊的时候，打打游戏总是需要的，游戏机要准备；坐在大巴士上，看着窗外美景，听听音乐应该也属于正常需求吧，MP3一定要带着了；有时或许还需要什么GPS来定定位，上网看看新闻，发发邮件，查查股票行情，这些需求如何办，总不能带着笔记本电脑在路上跑吧。这些东西且不说本身就很重，很麻烦，就说这些东西的充电器，就是五花八门，估计单就带这些东西，你就得累个半死了。”

大鸟：“你说得也没错，现在电子产品能玩的东西太多……”

小菜：“啊，大鸟！快看！”

3.2 拍摄

小菜惊呼，左手拉住大鸟的手，右手指向了天空。

大鸟跟着抬头一看，“那应该是架飞机吧！”

“不可能，”小菜坚决地说，“飞机哪有没翅膀的，那个东西飞得很奇怪，你看，你看，它停在空中，普通飞机怎么会在空中停下来。”

“是不太像飞机，飞碟？！——傻菜，快点用你手机录像呀！”

“是是是，啊，这手机怎么……等等，”小菜手忙脚乱。

“看你慌得，”大鸟说，“快些，马上可能就没了。”

“好了好了，”小菜终于打开了手机的摄像功能，对准了天空，“主要是对新功能不熟悉，你看，这家伙飞得多快。”

“嗯，它应该是飞碟，不然不可能这种样子的，以前也没有听说过这玩意，”大鸟肯定道，“还好你这手机可以摄像——啊，它飞跑了，你拍下来了没有？”

“好了，我都拍下来了，有点不太清楚，回去放电脑上看看吧。”小菜很开心，他的新手机发挥大作用了，“头一次看到UFO，就拍到了，这下可是大新闻了。”

“是呀，我也头一次看到，我们太幸运了。”

3.3　没用的东西

回到家中。小菜将手机文件传入电脑。

“这什么呀，黑乎乎的，什么也看不清。”大鸟大为失望。

“那不是有一个小白点吗？”小菜想极力申辩。

“那白点就和液晶显示器里的坏点一样，这如何看得出是UFO呢，说给别人谁信呀。”

“嗨！是的。”小菜也承认了这个事实，“这手机拍出来的东西没办法看呀，根本算不上是UFO的证据。”

小菜拿起手机，一脸苦相，对着它说道：“狗屁，要你这么多功能有鸟用，关键时刻就萎掉，我砸……”小菜举起手机欲往地上砸去。

3.4 单一职责原则

“砸呀，你砸呀！”大鸟笑嘻嘻地看着小菜，“哼哼，我就知道你舍不得，不过你的手机的确是太没用，这么好的机遇，都没有录成，如果是摄像机，效果一定不会差，因为当时我们眼睛看得很清楚呀。这下说给谁，谁也不信呀！大多数时候，一件产品简单一些，职责单一一些，或许是更好的选择。这就和设计模式中的一大原则——单一职责的道理是一样的。”

“哦，听字面意思，单一职责原则，意思就是说，功能要单一？”

“哈，可以简单地这么理解，它的准确解释是，就一个类而言，应该仅有一个引起它变化的原因 **[ASD]**。我们在做编程的时候，很自然地就会给一个类加各种各样的功能，比如我们写一个窗体应用程序，一般都会生成一个Form1这样的类，于是我们就把各种各样的代码，像某种商业运算的算法呀，像数据库访问的SQL语句呀什么的都写到这样的类当中，这就意味着，无论任何需求要来，你都需要更改这个窗体类，这其实是很糟糕的，维护麻烦，复用不可能，也缺乏灵活性。”

“是的，我写代码一般刚开始就是把所有的方法直接写在窗体类的代码当中的。”

> **单一职责原则（SRP）**，就一个类而言，应该仅有一个引起它变化的原因。

3.5　方块游戏的设计

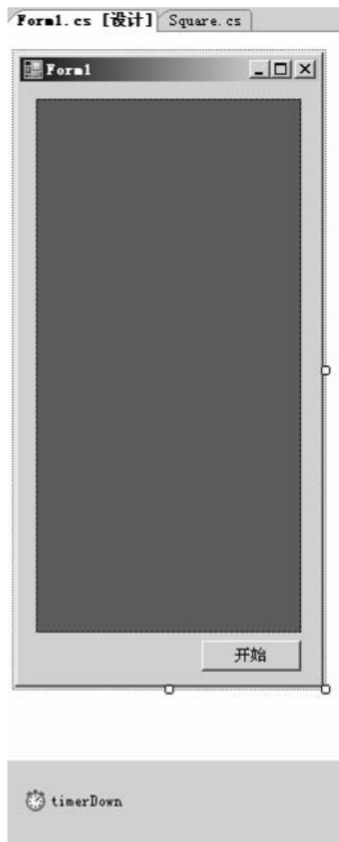
“我们再来举些例子，比如就拿手机里的俄罗斯方块游戏为例。要是让你开发这个小游戏，你如何考虑？”大鸟问道。

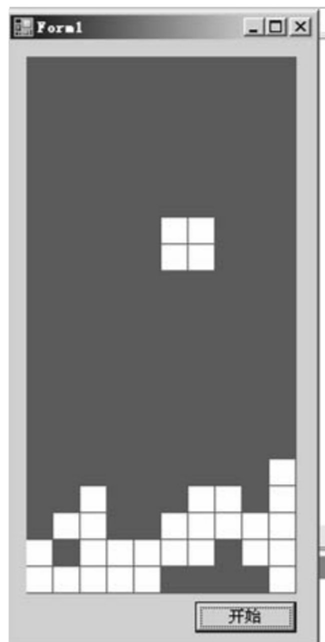
“我想想，首先它方块下落动画的原理是画四个小方块，擦掉，然后再在下一行画四个方块。不断地绘出和擦掉就形成了动画，所以应该要有画和擦方块的代码。然后左右键实现左移和右移，下键实现加速，上键实现旋转，这其实都应该是函数，当然左右移动需要考虑碰撞的问题，下移需要考虑堆积和消层的问题。”

“OK，你也说了不少了。如果就用WinForm的方式开发，你打算怎么开发呢？”

“那当然是先建立一个窗体Form，然后加一个用于游戏框的控件，比如Panel或者PictureBox，一个按钮Button来控制‘开始’，最后再放一个Timer控件用于分时动画的编程。写代码当然就是编写Timer_Tick事件来绘出和擦除方块，并做出堆积和消层的判断。再编写控件的键盘事件，按了左箭头则左移，右箭头则右移等等。对了，还需要用到些GDI＋技术的方法来画方块和擦方块。”

“你能不能就这些代码划分一下类呢？”





“分类？这里好像关键在于各种事件代码如何写吧，这里有什么类可言呢？”

“看来你的面向过程开发已经根深蒂固了。你把所有的代码都写在了Form1.cs这个类里，你觉得这合理吗？”

“可能不合理，但我实在没想出怎么分离它。”

“打个比方，如果现在要你写的是手机版的俄罗斯方块程序，即Pocket PC或者Windows CE上运行的程序，它们可以安装.NET框架的精简版，运行C#语言编写的应用程序，但PC上的普通WinForm界面的程序不能使用。那你现在这个代码有什么可以复用的吗？”

“你都已经说了，不能使用，我当然就没法使用了。Copy过去，再针对代码做些改进吧。”

“但这当中，有些东西是始终没变的。”

“你是说，下落、旋转、碰撞判断、移动、堆积这些游戏逻辑吧？”

“说得没错，这些都是和游戏有关的逻辑，和界面如何表示没有什么关系，为什么要写在一个类里面呢？如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受到意想不到的破坏 **[ASD]** 。事实上，你完全可以找出哪些是界面，哪些是游戏逻辑，然后进行分离。”

“但我还是不明白，如何分离开。”

“你仔细想想看，方块的可移动的游戏区域，可以设计为一个二维整型数组用来表示坐标，宽10，高20，比如‘int[，]　arraySquare＝new int[10，20];’，那么整个方块的移动其实就是数组的下标变化，比如原方块在arraySquare [3，5]上，则下移时变成arraySquare [3，6]，如果下移同时还按了左键，则是arraySquare　[2，6]。每个数组的值就是是否存在方块的标志，存在为1，不存在时缺省为0。这下你该明白，所谓的碰撞判断，其实就是什么？”

“我知道了，是否能左移，就是判断arraySquare　[x，y]中的x–1是否小于0，否则就撞墙了。或者arraySquare　[x–1，y]是否等于1，否则就说明左侧有堆积的方块。所谓堆积，不过是判断arraySquare　[x，y＋1]是否等于1的过程，如果是，则将自己arraySquare　[x，y]的值改1。那么消层，其实就是arraySquare　[x，y]中循环x由0到9，判断arraySquare [x，y]是否都等于1，是则此行数据清零，并将其上方的数组值遍历下移一位。”

“那你就应该明白了，所谓游戏逻辑，不过就是数组的每一项值变化的问题，下落、旋转、碰撞判断、移动、堆积这些都是在做数组具体项的值的变化。而界面表示逻辑，不过是根据数组的数据进行绘出和擦除，或者根据键盘命令调用数组的相应方法进行改变。因此，至少应该考虑将此程序分为两个类，一个是游戏逻辑的类，一个是WinForm窗体的类。当有一天要改变界面，或者换界面时，不过是窗体类的变化，和游戏逻辑无关，以此达到复用的目的。”

“这个听起来容易，真正要做起来还是有难度的哦！”

“当然，软件设计真正要做的许多内容，就是发现职责并把那些职责相互分离 **[ASD]** 。其实要去判断是否应该分离出类来，也不难，那就是如果你能够想到多于一个的动机去改变一个类，那么这个类就具有多于一个的职责 **[ASD]** ，就应该考虑类的职责分离。”

“的确是这样，界面的变化是和游戏本身没有关系的，界面是容易变化的，而游戏逻辑是不太容易变化的，将它们分离开有利于界面的改动。”

3.6 手机职责过多吗？

“这下你知道你的手机为什么不能拍摄好UFO的原因了吧？”大鸟笑道。

“如果手机只用来接听电话，DV用来拍摄，职责的分离是可以把事情做得更好。不过这其实不是一回事哦，现在的智能手机承担的职责多，并不等于就不可以做好，只不过现在的科技还不能让手机在摄像时超过DV而已。”小菜分析说。

“整合当然是一种很好的思想。比如Google最初的理想就是将一切的需求都整合到一个文本框里提交，用干净的页面来吸引用户，导致互联网的一场变革。但现在分类信息、垂直搜索又开始流行，这却是单一职责的思想体现。现在智能手机整合了很多功能的原因是因为DV、DC、MP3等产品的体积也太大了。手机携带很方便，所以才有了这样的过渡产品，如果，每一样数码产品都缩小100倍，就像放在包里的一张卡片、一支笔那么简单，而功能和质量都不发生变化，你还会觉得它们很麻烦吗？”大鸟总结道，“总的来说，手机的发展有它的特点，而编程时，我们却是要在类的职责分离上多思考，做到单一职责，这样你的代码才是真正的易维护、易扩展、易复用、灵活多样。”

第4章　考研求职两不误——开放-封闭原则

4.1　考研失败

时间：3月5日20点　　地点：小菜房间　　人物：小菜、大鸟

“……多少次迎着冷眼与嘲笑，从没有放弃过心中的理想，一刹那恍惚，若有所失的感觉，不知不觉已变淡心里爱（谁明白我）……”

小菜此时正关在房中坐在桌前发呆，音箱中大声地放着Beyond乐队的《海阔天空》。此时有人敲门。打开一看，原来是大鸟。

大鸟：“小菜，怎么听这么伤感的歌，声音这么大，我在隔壁都听得清清楚楚。发生什么事了？”

小菜：“今天研究生考试成绩出来了，我的英语成绩离分数线差两分。之前的努力白费了。”

大鸟：“失败也是正常的，考不上的人还是占多数呀，想开些吧，找到好工作未必比读研要差的。”

小菜：“为了考研，我没有做任何求职的准备，所以我们班不少同学都找到工作了，我却才刚开始，前段时间的面试也没消息。”

大鸟：“哈，鱼和熊掌岂能兼得，为一件事而放弃另一些机会，也是在情理之中的事。”

小菜：“说是这么说，我却感觉比较难受，我的同学，有几个其实水平不比我强，他们都签了XX大集团、XX知名公司，而我现在一无所有，感觉很糟糕。要是当时我也花点时间在简历上，或许现在也不至于这么不爽。”

大鸟：“你考研复习的时候，每天学习多长时间，有没有休息的时候？”

小菜：“差不多十小时吧，其实效率并不高，有不少时候都困得不行，趴在桌上睡觉去了。”

大鸟："这就对了，你为什么不利用休息的时间考虑一下自己的简历如何写，关心一下有些什么单位在招聘呢？这样也就不至于现在这样唉声叹气。"

小菜："我感觉找工作会影响复习的精力，所以干脆什么都没找，但其实每天都会有些同学求职应聘的消息传到我耳朵里，我也没有安心复习。"

大鸟："小菜呀，你其实就是没有搞懂一个设计模式的原则。"

小菜："哦，是什么原则？"

大鸟："先不谈这个原则，你想想看，香港澳门的顺利回归，有一个人起了重要的作用，他是谁？"

小菜："啊，那还用说，是邓小平呀，如果不是他老人家提出的一国两制思想，或许现在还没回归呢。"

大鸟："小平同志的确是伟大的政治思想家，他的这一创造性想法有什么独到之处？"

小菜："我想想看，原因主要是在于大陆的社会主义制度不能修改，这一点毋庸置疑，而香港澳门长期在资本主义制度下管理和发展，所以回归时强行修改香港澳门的制度也并不合理，所以用'一国两制'来解决制度差异造成的矛盾是最合理的办法。"

大鸟："说得好，社会主义制度不能修改，邓小平在和英国首相撒切尔夫人谈香港问题的时候，如果咬定香港回来必须要实现社会主义制度，那回归就困难重重了，香港老百姓也不答应呀，毕竟这么多年来的殖民统治，突然在整个管理制度上进行彻底变化也是不现实的，那么怎么办？为了回归的大局，增加一种制度又何尝不可，一个国家，两种制度，这在政治上，是伟大的发明哦。在软件设计模式中，这种不能修改，但可以扩展的思想也是最重要的一种设计原则，它就是开放-封闭原则（The Open-Closeed Principle，简称OCP）或叫开-闭原则。"

4.2 开放-封闭原则

小菜："开放、封闭，具体怎么解释呢？"

> **开放-封闭原则**，是说软件实体（类、模块、函数等等）应该可以扩展，但是不可修改。[ASD]

大鸟："这个原则其实是有两个特征，一个是说'对于扩展是开放的（**Open for extension**）'，另一个是说'对于更改是封闭的（**Closed for modification**）'**[ASD]**。"

大鸟："我们在做任何系统的时候，都不要指望系统一开始时需求确定，就再也不会变化，这是不现实也不科学的想法，而既然需求是一定会变化的，那么如何在面对需求的变化时，设计的软件可以相对容易修改，不至于说，新需求一来，就要把整个程序推倒重来。怎样的设计才能面对需求的改变却可以保持相对稳定，从而使得系统可以在第一个版本以后不断推出新的版本呢？**[ASD]**，开放-封闭给我们答案。"

小菜："我明白了，你的意思是说，设计软件要容易维护又不容易出问题的最好的办法，就是多扩展，少修改？"

大鸟："是的，比如说，我是公司老板，我规定，九点上班，不允许迟到。但有几个公司骨干，老是迟到。如果你是老板你怎么做？"

小菜："严格执行考勤制度，迟到扣钱。"

大鸟："你倒是够狠，但实际情况是，有的员工家离公司太远，有的员工每天上午要送小孩子上学，交通一堵就不得不迟到了。"

小菜："这个，让他们有特殊原因的人打报告，然后允许他们迟到。"

大鸟："哈，谈何容易，别的不迟到的员工不答应了呀，凭什么他能迟到，我就不能，大家都是工作，我上午也完全可以多睡会再来。"

小菜："那怎么办？老是迟到的确也不好，但不让迟到也不现实。家的远近，交通是否堵塞也不是可以控制的。"

大鸟："仔细想想，你会发现，其实迟到不是主要问题，每天保证8小时的工作量是老板最需要的，甚至8小时工作时间也不是主要问题，业绩目标的完成或超额完成才是最重要的指标，于是应该改变管理方式，比如弹性上班工作制，早到早下班，晚到晚下班，或者每人每月允

许三次迟到，迟到者当天下班补时间等等，对市场销售人员可能就更加以业绩为标准，工作时间不固定了——这其实就是对工作时间或业绩成效的修改关闭，而对时间制度扩展的开放。”

小菜：“这就需要老板自己很清楚最希望达到的目的是什么，制定的制度才最合理有效。”

大鸟：“对的，用我们古人的理论来说，管理需要中庸之道。”

4.3 何时应对变化

小菜："啊，有道理。所以，我们尽量应在设计时，考虑到需求的种种变化，把问题想得全了，就不会因为需求一来，手足无措。"

大鸟："哪有那么容易，如果什么问题都考虑得到，那不就成了未卜先知，这是不可能的。需求时常会在你想不到的地方出现，让你防不胜防。"

小菜："那我们应该怎么做？"

大鸟："开放-封闭原则的意思就是说，你设计的时候，时刻要考虑，尽量让这个类是足够好，写好了就不要去修改了，如果新需求来，我们增加一些类就完事了，原来的代码能不动则不动。"

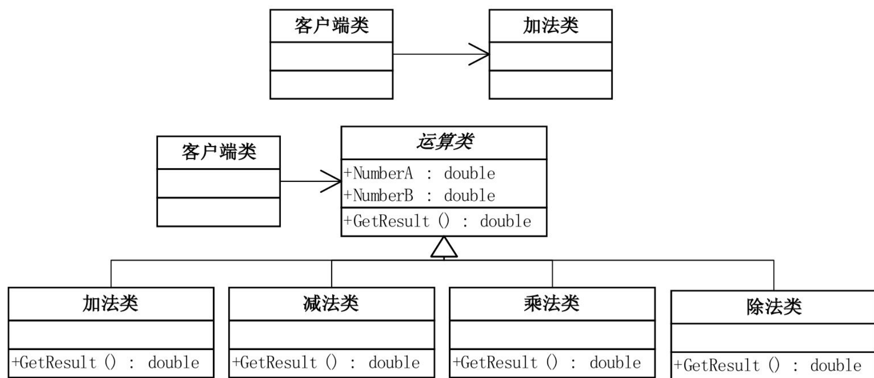
小菜："这可能做到吗？我深表怀疑呀，怎么可能写完一个类就再也不改了呢？"

大鸟："你说得没错，绝对的对修改关闭是不可能的。无论模块是多么的'封闭'，都会存在一些无法对之封闭的变化。既然不可能完全封闭，设计人员必须对于他设计的模块应该对哪种变化封闭做出选择。他必须先猜测出最有可能发生的变化种类，然后构造抽象来隔离那些变化**[ASD]**。"

小菜："那还是需要猜测程序可能会发生的变化，猜对了，那是成功，猜错了，那就完全走到另一面去了，把本该简单的设计做得非常复杂，很不划算呀。而且事先猜测，这又是很难做到的。"

大鸟："你说得没错，我们是很难预先猜测，但我们却可以在发生小变化时，就及早去想办法应对发生更大变化的可能。也就是说，等到变化发生时立即采取行动**[ASD]**。正所谓，同一地方，摔第一跤不是你的错，再次在此摔跤就是你的不对了。"

大鸟："在我们最初编写代码时，假设变化不会发生。当变化发生时，我们就创建抽象来隔离以后发生的同类变化**[ASD]**。比如，我之前让你写的加法程序，你很快在一个client类中就完成，此时变化还没有发生。然后我让你加一个减法功能，你发现，增加功能需要修改原来这个类，这就违背了今天讲到的'开放-封闭原则'，于是你就该考虑重构程序，增加一个抽象的运算类，通过一些面向对象的手段，如继承，多态等来隔离具体加法、减法与client耦合，需求依然可以满足，还能应对变化。这时我又要你再加乘除法功能，你就不需要再去更改client以及加法减法的类了，而是增加乘法和除法子类就可。即面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码**[ASD]**。这就是'开放-封闭原则'的精神所在。"（样例代码见第1章）



大鸟："当然，并不是什么时候应对变化都是容易的。我们希望的是在开发工作展开不久就知道可能发生的变化。查明可能发生的变化所等待的时间越长，要创建正确的抽象就越困难 **[ASD]** 。"

小菜："这个我能理解，如果加减运算都在很多地方应用了，再考虑抽象、考虑分离，就很困难。"

大鸟："开放-封闭原则是面向对象设计的核心所在。遵循这个原则可以带来面向对象技术所声称的巨大好处，也就是可维护、可扩展、可复用、灵活性好。开发人员应该仅对程序中呈现出频繁变化的那些部分做出抽象，然而，对于应用程序中的每个部分都刻意地进行抽象同样不是一个好主意。拒绝不成熟的抽象和抽象本身一样重要 **[ASD]** 。切记，切记。"

小菜："哦，我还以为尽量地抽象是好事呢，看来过犹不及呀。"

4.4　两手准备，并全力以赴

大鸟："回过头来说，你考研和求职这两件事，考研是你的追求，希望考上研究生，可以更上一层楼，有更大的发展空间和机会。所以考研之前，学习计划是不应该更改，雷打不动的。这就是对修改关闭。但你要知道，你几个月来只埋头学习，就等于放弃了许多好公司来你们学校招聘的机会，这机会的失去是很不值得的。我就不信你一天到晚全在学习，那样效果也不会好。所以你完全可以抽出一点时间，在不影响你复习的前提下，来写写自己的简历，来了解一些招聘大学生的公司的资讯，这不是很好的事吗？既不影响你考研，又可以增大找到好工作的可能性。为考研万一失败后找工作做好了充分的准备。这就是对扩展开放，对修改关闭的意义。"

小菜："是的，我就不信，我会比别人差！"

大鸟笑了笑说："好了，我回房间去了，你也早些休息吧。"站起身走出了小菜的房门，此时Beyond的音乐再次响起，大鸟回头，伸出右手向前摆了个"V"字，说了声："海阔天空，加油！"

"今天我寒夜里看雪飘过，怀着冷却了的心窝漂远方，风雨里追赶，雾里分不清影踪，天空海阔你与我可会变（谁没在变），…………仍然自由自我，永远高唱我歌，走遍千里！"（作者注：本故事和"开放-封闭原则"对应有些牵强，所以在此做一声明。全力以赴当然是必需，两手准备也是灵活处事的表现，希望读者您能对痛苦关闭，对快乐开放。）

第5章　会修电脑不会修收音机？——依赖倒转原则

5.1　MM请求修电脑

时间：3月12日　19点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟、娇娇

小菜和大鸟吃完晚饭后，在一起聊天。

此时，突然声音响起。

“死了都要爱，不淋漓尽致不痛快，感情多深只有这样，才足够表白。死了都要爱……”

原来是小菜的手机铃声，大鸟吓了一跳，说道：“你小子，用这歌做铃声，吓唬人啊！这要是在公司开大会时响起，你要被领导淋漓尽致爱死！MD，还在唱，快接！”

小菜很是郁闷，拿起手机一看，一个美女来的电话，脸色由阴转晴，马上接通了手机：“喂！”

“小菜呀，我是娇娇，我电脑坏了，你快点帮帮我呀！”手机里传来急促的女孩声音。

“哈，是你呀，你现在好吗？最近怎么不和我聊天了？”小菜慢条斯理地说道。

“快点帮帮我呀，电脑不能用了啊！”娇娇略带哭腔地说。

“别急别急，怎么个坏法？”

“每次打开QQ，一玩游戏，机器就死了。出来蓝底白字的一堆莫名其妙的英文，过一会就重启了，再用QQ还是一样。怎么办呀？”

“哦，明白了，蓝屏死机吧，估计内存有问题，你的内存是多少兆的？”

“什么内存多少兆，我听不懂呀，你能过来帮我修一下吗？”

“啊，你在金山，我在宝山，虽说在上海这两地名都钱味儿十足，可两山相隔万重路呀！现在都晚上了，又是星期一，周六我去你那里帮你修吧！”小菜无奈地说。

“要等五天那不行，你说什么蓝屏？怎么个修法？”娇娇依然急不可待。

“蓝屏多半是内存坏了，你要不打开机箱看看，或许有两个内存，可以拔一根试试，如果只有一根内存，那就没戏了。”

“机箱怎么打开呢？”娇娇开始认真起来。

“这个，你找机箱后面，四个角应该都有螺丝，卸掉靠左侧边上两个应该就可以打开左边盖了。”小菜感觉有些费力，远程手机遥控修电脑，这是头一次。

“我好像看到了，要不先挂电话，我试试看，打开后再打给你。”

“哦，好的。”小菜正说着，只听娇娇边嘟囔着“老娘就不信收拾不了你这破电脑”边挂掉了电话。

“呵！”小菜长出一口气，“不懂内存为何物的美眉修电脑，强！”

“你小子，人家在困难时刻想得到你，说明心中有你，懂吗？这是机会！”大鸟说道。

“这倒也是，这小美眉长得蛮漂亮的，我看过照片。就是脾气大些，不知道有没有男朋友了。”

“切，你干吗不对她说，‘你可以找男友修呀’，真是没脑子，要是有男友，就算男友不会修也要男友找人搞定，用得着找你求助呀，笨笨！”大鸟嘲笑道，“你快把你那该死的手机铃声换掉——死了都要爱，死了还爱个屁！”

“噢！知道了。”

5.2 电话遥控修电脑

十分钟后。

“我在这儿等着你回来，等着你回来，看那桃花开。我在这儿等着你回来，等着你回来，把那花儿采……”小菜的手机铃声再次响起。

“菜花痴，你就不能找个好听的歌呀。”大鸟气着说道。

“好好好，我一会改，一会改。”小菜拿起手机，一副很听话的样子，嘴里却跟着哼“我在这儿等着你回来哎”，把手机放到耳边。

“小菜，我打开机箱了，快说下一步怎么走！”娇娇仍然着急着说。

“你试着找找内存条，大约是10公分长，2公分宽，上有多个小长方形集成电路块的长条，应该是竖插着的。”小菜努力把内存条的样子描述得容易理解。

“我看到一个风扇，没有呀，在哪里？”娇娇说道，“哦，我找到了，是不是很薄，很短的小长条？咦，怎么有两根？”

“啊，太好了，有两根估计就能解决问题了，你先试着拔一根，然后开机试试看，如果还是死机，再插上，拔另一根试，应该总有一根可以保证不蓝屏。”

“我怎么拨不下来呢？”

“旁边有卡子，你扳开再试。”

“嗯，这下好了，你别挂，我这就重启看看。”

五分钟后。

“哈，没有死机了啊，小菜，你太厉害了，我竟然可以修电脑了，要我怎么感谢你呢！”娇娇兴奋 地说。

“做我女朋友吧，”小菜心里这么遐想着，口中却谦虚地说：“不客气，都是你聪明，敢自己独自打开机箱修电脑的女孩很少的。你把换下的内存去电脑城换掉，就可以了。”

“我不懂的，要不周六你帮我换？周六我请你吃饭吧！”

“这怎么好意思——你说在什么时间在哪碰面？”小菜假客气着，却不愿意放弃机会。

“周六下午5点在徐家汇太平洋数码门口吧。”

“好的，没问题。”

“今天真的谢谢你，那就先Bye-Bye了！”

“嗯，拜拜！”

5.3 依赖倒转原则

“小菜走桃花运了哦，”大鸟有些羡慕道，“那铃声看来有些效果，不过还是换掉吧，俗！”

“嘿嘿，你说也怪，修电脑，这在以前根本不可能的事，怎么就可以通过电话就教会了，而且是真的修到可以用了呢。”

“你有没有想过这里的最大原因？”大鸟开始上课了。

“蓝屏通常是内存本身有问题或内存与主板不兼容，主板不容易换，但内存更换起来很容易。”

“如果是别的部件坏了，比如硬盘，显卡，光驱等，是否也只需要更换就可以了？”

“是呀，确实很方便，只需要懂一点点计算机知识，就可以试着修电脑了。”

“想想这和我们编程有什么联系？”

“你的意思又是——面向对象？”

“嗯，说说看，面向对象的四个好处？”

“这个我记得最牢了，就是活字印刷那个例子呗。是可维护、可扩展、可复用和灵活性好。哦，我知道了，可以把PC电脑理解成是大的软件系统，任何部件如CPU、内存、硬盘、显卡等都可以理解为程序中封装的类或程序集，由于PC易插拔的方式，那么不管哪一个出问题，都可以在不影响别的部件的前提下进行修改或替换。”

“PC电脑里叫易插拔，面向对象里把这种关系叫什么？”

“应该是叫强内聚、松耦合吧。”

“对的，非常好，我们电脑里的CPU全世界也就是那么几家生产的，大家都在用，但却不知道Intel、AMD等公司是如何做出这个精密的小东西的。去年国内不是还出现了汉芯造假的新闻吗！这就说明CPU的强内聚的确是强。但它又独自成为了产品，在千千万万的电脑主板上插上就可以使用，这是什么原因？”大鸟又问。

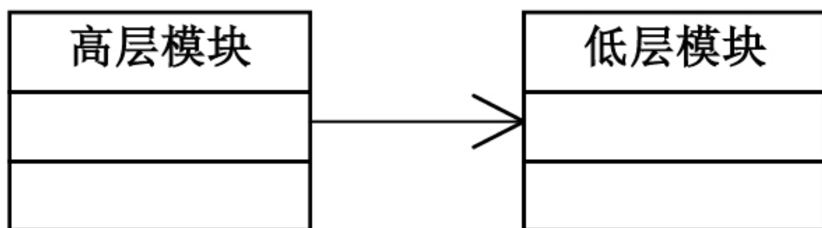
“因为CPU的对外都是针脚式或触点式等标准的接口。啊，我明白了，这就是接口的最大好处。CPU只需要把接口定义好，内部再复杂我也不让外界知道，而主板只需要预留与CPU针脚的插槽就可以了。”

“很好，你已经在无意的谈话间提到了面向对象的几大设计原则，比如我们之前讲过的单一职责原则，就刚才修电脑的事，显然内存坏了，不应该成为更换CPU的理由，它们各自的职责是明确的。再比如开放-封闭原则，内存不够只要插槽足够就可以添加，硬盘不够可以用移动硬盘等，PC的接口是有限的，所以扩展有限，软件系统设计得好，却可以无限地扩展。这两个原则我们之前都已经提过了。这里需要重点讲讲一个新的原则，叫依赖倒转原则，也有翻译成依赖倒置原则的。”大鸟接着讲道，“依赖倒转原则，原话解释是抽象不应该依赖细节，细节应该依赖于抽象，这话绕口，说白了，就是要针对接口编程，不要对实现编程，无论主板、CPU、内存、硬盘都是在针对接口设计的，如果针对实现来设计，内存就要对应到具体的某个品牌的主板，那就会出现换内存需要把主板也换了的尴尬。你想在小MM面前表现也就不那么容易了。所以说，PC电脑硬件的发展，和面向对象思想发展是完全类似的。这也说明世间万物都是遵循某种类似的规律，谁先把握了这些规律，谁就最早成为了强者。”

> 依赖倒转原则
>
> A．高层模块不应该依赖低层模块。两个都应该依赖抽象。
>
> B．抽象不应该依赖细节。细节应该依赖抽象。[ASD]

“为什么要叫倒转呢？”小菜问道。

“这里面是需要好好解释一下，面向过程的开发时，为了使得常用代码可以复用，一般都会把这些常用代码写成许许多多函数的程序库，这样我们在做新项目时，去调用这些低层的函数就可以了。比如我们做的项目大多要访问数据库，所以我们就把访问数据库的代码写成了函数，每次做新项目时就去调用这些函数。这也就叫做高层模块依赖低层模块。”



“嗯，是这样的，我以前都是这么做的。这有什么问题？”

“问题也就出在这里，我们要做新项目时，发现业务逻辑的高层模块都是一样的，但客户却希望使用不同的数据库或存储信息方式，这时就出现麻烦了。我们希望能再次利用这些高层模块，但高层模块都是与低

层的访问数据库绑定在一起的，没办法复用这些高层模块，这就非常糟糕了。就像刚才说的，PC里如果CPU、内存、硬盘都需要依赖具体的主板，主板一坏，所有的部件就都没用了，这显然不合理。反过来，如果内存坏了，也不应该造成其他部件不能用才对。而如果不管高层模块还是低层模块，它们都依赖于抽象，具体一点就是接口或抽象类，只要接口是稳定的，那么任何一个的更改都不用担心其他受到影响，这就使得无论高层模块还是低层模块都可以很容易地被复用。这才是最好的办法。”

“为什么依赖了抽象的接口或抽象类，就不怕更改呢？”小菜依然疑惑，“不好意思，我有些钻牛角尖了。”

“没有，没有，在这里弄不懂是很正常的，原因是我少讲了一个设计原则，使得你产生了困惑，这个原则就是里氏代换原则。”

5.4 里氏代换原则

“里氏代换原则是Barbara Liskov女士在1988年发表的 [ASD]，具体的数学定义比较复杂，你可以查相关资料，它的白话翻译就是一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且它察觉不出父类对象和子类对象的区别。也就是说，在软件里面，把父类都替换成它的子类，程序的行为没有变化，简单地说，子类型必须能够替换掉它们的父类型[ASD]。”

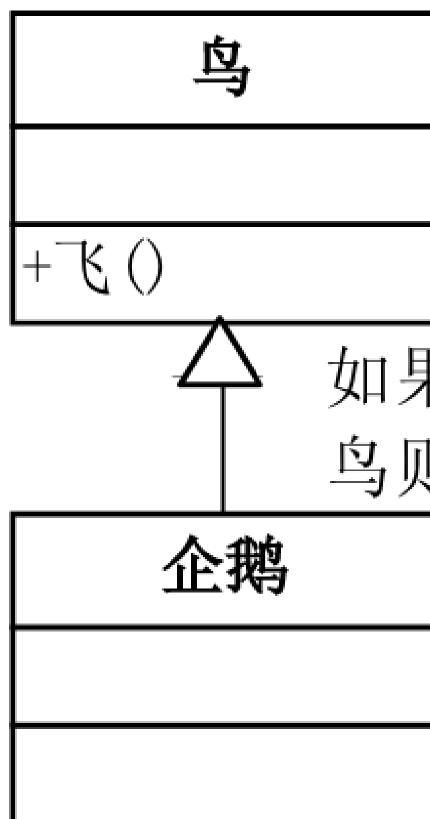
里氏代换原则（**LSP**）：子类型必须能够替换掉它们的父类型。[ASD]

“这好像是学继承时就要理解的概念，子类继承了父类，所以子类可以以父类的身份出现。”

“是的，我问你个问题，如果在面向对象设计时，一个是鸟类，一个是企鹅类，如果鸟是可以飞的，企鹅不会飞，那么企鹅是鸟吗？企鹅可以继承鸟这个类吗”

“企鹅是一种特殊的鸟，尽管不能飞，但它也是鸟呀，当然可以继承。”

“哈，你上当了，我说的是在面向对象设计时，那就意味着什么呢？子类拥有父类所有非private的行为和属性。鸟会飞，而企鹅不会飞。尽管在生物学分类上，企鹅是一种鸟，但在编程世界里，企鹅不能以父类——鸟的身份出现，因为前提说所有鸟都能飞，而企鹅飞不了，所以，企鹅不能继承鸟类。”

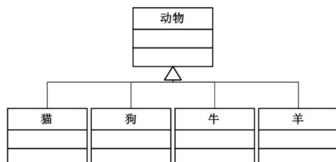


“哦，你的意思我明白了，我受了直觉的影响。小时候上课时老师一再强调，像鸵鸟、企鹅等不会飞的动物也是鸟类。”

“也正因为有了这个原则，使得继承复用成为了可能，只有当子类可以替换掉父类，软件单位的功能不受到影响时，父类才能真正被复用，而子类也能够在父类的基础上增加新的行为。比方说，猫是继承动物类的，以动物的身份拥有吃、喝、跑、叫等行为，可当某一天，我们需要狗、牛、羊也拥有类似的行为，由于它们都是继承于动物，所以除了更改实例化的地方，程序其他处不需要改变。”



扩展

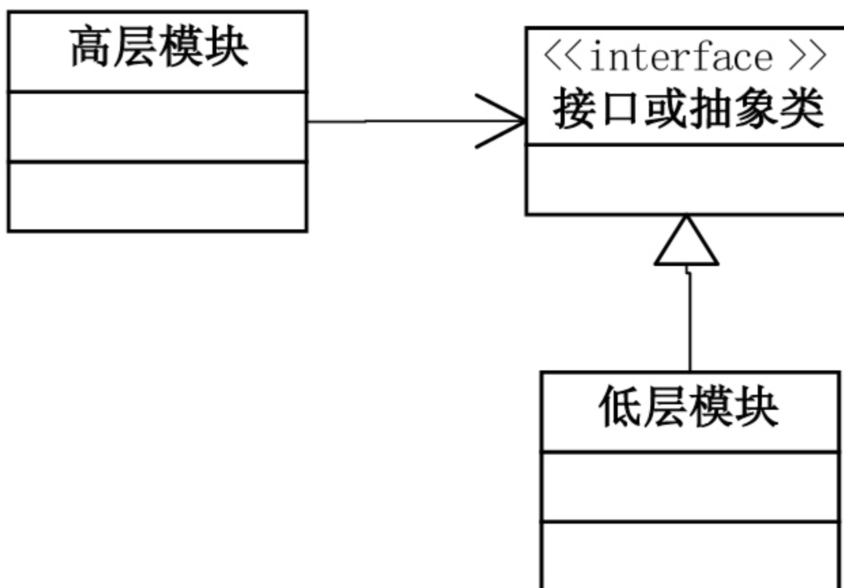



```
动物 animal = new 猫();
animal.吃();
animal.喝();
animal.跑();
animal.叫();
```

需求的变化，使得需要将“猫”更换成“狗”、“牛”、“羊”等别的动物，程序其他地方不需要改变

“我的感觉，由于有里氏代换原则，才使得开放-封闭成为了可能。”小菜说。

“这样说是可以的，正是由于子类型的可替换性才使得使用父类类型的模块在无需修改的情况下就可以扩展。不然还谈什么扩展开放，修改关闭呢。再回过头来看依赖倒转原则，高层模块不应该依赖低层模块，两个都应该依赖抽象，对这句话你就会有更深入的理解了。”



“哦，我明白了，依赖倒转其实就是谁也不要依靠谁，除了约定的接口，大家都可以灵活自如。还好，她没有问我如何修收音机，收音机里都是些电阻、三极管，电路板等等东东，全都焊接在一起，我可不会修的。”小菜庆幸道。

5.5 修收音机

“哈，小菜你这个比方打得好，”大鸟开心地说，“收音机就是典型的耦合过度，只要收音机出故障，不管是没有声音、不能调频，还是有杂音，反正都很难修理，不懂的人根本没法修，因为任何问题都可能涉及其他部件，各个部件相互依赖，难以维护。非常复杂的PC电脑可以修，反而相对简单的收音机不能修，这其实就说明了很大的问题。当然，电脑的所谓修也就是更换配件，CPU或内存要是坏了，老百姓是没法修的。现在在软件世界里，收音机式的强耦合开发还是太多了，比如前段时间某银行出问题，需要服务器停机大半天的排查修整，这要损失多少钱。如果完全面向对象的设计，或许问题的查找和修改就容易得多。依赖倒转其实可以说是面向对象设计的标志，用哪种语言来编写程序不重要，如果编写时考虑的都是如何针对抽象编程而不是针对细节编程，即程序中所有的依赖关系都是终止于抽象类或者接口，那就是面向对象的设计，反之那就是过程化的设计了 **[ASD]** 。”

“是的是的，我听说很多银行目前还是纯C语言的面向过程开发，非常不灵活，维护成本是很高昂的。”

“那也是没办法的，银行系统哪是说换就换的，所以现在是大力鼓励年轻人学设计模式，直接面向对象的设计和编程，从大的方向上讲，这是国家大力发展生产力的很大保障呀。”

“大鸟真是高瞻远瞩呀，我对你的敬仰犹如滔滔江水，连绵不绝！”小菜怪笑道，“我去趟WC”。

“浪奔，浪流，万里江海点点星光耀，人间事，多纷扰，化作滚滚东逝波涛，有泪，有笑…………”

“小菜，电话。小子，怎么又换成新上海滩的歌了，这歌好听。”大鸟笑道，“刚才是死了都要爱，现在是为爱复仇而死。你怎么找的歌都跟爱过不去呀。快点，电话，又是刚才那个叫娇娇的小MM的。”

“来了来了，尿都只尿了一半！”小菜心急地接起电话，“喂！”

“小菜呀，我家收音机坏了，你能不能教我修修呢！”

第6章　穿什么有这么重要？——装饰模式

6.1　穿什么有这么重要？

时间：3月16日20点　　地点：大鸟房间　　人物：小菜、大鸟

“大鸟，明天我要去见娇娇了，你说我穿什么去比较好？”小菜问大鸟道。

“这个你也来问我，干脆我代你去得了。”大鸟笑言。

“别开玩笑，我是诚心问你的。”

“哈哈，小菜呀，你别告诉我说四年大学你都没和女生约过会？”

“嗨！谁叫我念的是理工科大学呢，学校里本来女生就少，所以这些稀有宝贝们，大一时早就被那些老手们主动出击搞定了，我们这些恋爱方面的小菜哪还有什么机会，一不小心就虚度了四年。”小菜突生伤感，叹气摇头，并小声的唱了起来，“不是我不小心，只是真情难以寻觅，不是我存心故意，只因无法找到良机……”

“喂！”大鸟打断了小菜，“差不多就行了，感慨得没完没了。说正事，你要问我什么？”

“哦，你说我穿什么去见娇娇比较好？”

“那要看你想给人家什么印象？是比较年轻，还是比较干练；是比较颓废，还是要比较阳光；也有可能你想给人家一种极其难忘的印象，那穿法又大不一样了！”

“你这话怎么讲？”

“年轻，不妨走点嘻哈路线，大T恤、垮裤、破球鞋，典型的年轻人装扮。”

“啊，这不是我喜欢的风格，我从来也没这样穿过。”

“那就换一种，所谓干练，就是要有外企高级白领的样，黑西装、黑

领带、黑墨镜、黑皮鞋……”

“你这叫白领？我看是黑社会。不行不行。”

“哈，来颓废的吧，颓废其实也是一种个性，可以吸引一些喜欢叛逆的女生。一般来说，其标志是：头发可养鸟、胡子能生虫、衬衣没纽扣、香烟加狐臭。”

“这根本就是‘肮脏’的代表吗，开什么玩笑。你刚才提到给人家难忘印象，是什么样的穿法？”

“哈，这当然是绝妙的招了，如果你照我说的去做，娇娇想忘都难。”

“快说快说，是什么？”

“大红色披风下一身蓝色紧身衣，胸前一个大大的‘S’，表明你其实穿的是‘小号’，还有最重要的是，一定要‘内裤外穿’……”

“喂，你拿我寻开心呀，那是‘超人’的打扮呀，‘S’代表的也不是‘Small’，是‘Super’的意思。”小菜再次打断了大鸟，还是忍不住笑道，“我如果真的穿这样的服装去见MM，那可真是现场的人都终身难忘，而小菜我，估计这辈子也不要见人了。”

“哈，你终于明白了！我其实想表达的意思就是，你完全可以随便一些，平时穿什么，明天还是穿什么，男生嘛，只要干净一些就可以了，关键不在于你穿什么，而在于你人怎么样。对自己都这么没信心，如何追求女孩子。”

“哦，我可能是多虑了一些。”小菜点头道，“好吧，穿什么我自己再想想。没什么事了吧，那我回房了。”

“等等，”大鸟叫住了他，“今天的模式还没有开讲呢，怎么就跑了。”

“哦，我想着约会的事，把学习给忘了，今天学什么模式？”

6.2 小菜扮靓第一版

“先不谈模式，说说你刚才提到的穿衣问题。我现在要求你写一个可以给人搭配不同的服饰的系统，比如类似QQ、网络游戏或论坛都有的Avatar系统。你怎么开发？”

“你是说那种可以换各种各样的衣服裤子的个人形象系统？”

“是的，现在你就简单点，用控制台的程序，写可以给人搭配嘻哈服或白领装的代码。”

“哦，我试试看吧。”

半小时后，小菜的第一版代码出炉。

结构图

人

+穿大T恤()
+穿垮裤()
+穿破球鞋()
+穿西装()
+打领带()
+穿皮鞋()
+形象展示()

“Person”类

```
class Person
{
    private string name;
    public Person(string name)
    {
        this.name = name;
    }
    public void WearTShirts()
    {
        Console.Write("大T恤 ");
    }
    public void WearBigTrouser()
    {
        Console.Write("垮裤 ");
    }
    public void WearSneakers()
    {
        Console.Write("破球鞋 ");
    }
    public void WearSuit()
    {
        Console.Write("西装 ");
    }
    public void WearTie()
    {
        Console.Write("领带 ");
    }
    public void WearLeatherShoes()
    {
        Console.Write("皮鞋 ");
    }
    public void Show()
    {
        Console.WriteLine("装扮的{0}",name);
    }
}
```

客户端代码

```
static void Main(string[] args)
{
```

```
        Person xc = new Person("小菜");

        Console.WriteLine("\n第一种装扮：");

        xc.WearTShirts();
        xc.WearBigTrouser();
        xc.WearSneakers();
        xc.Show();

        Console.WriteLine("\n第二种装扮：");

        xc.WearSuit();
        xc.WearTie();
        xc.WearLeatherShoes();
        xc.Show();

        Console.Read();
    }
```

结果显示

```
第一种装扮：
大T恤  垮裤  破球鞋  装扮的小菜

第二种装扮：
西装  领带  皮鞋  装扮的小菜
```

“哈，不错，功能是实现了。现在的问题就是如果我需要增加‘超人’的装扮，你得如何做？”

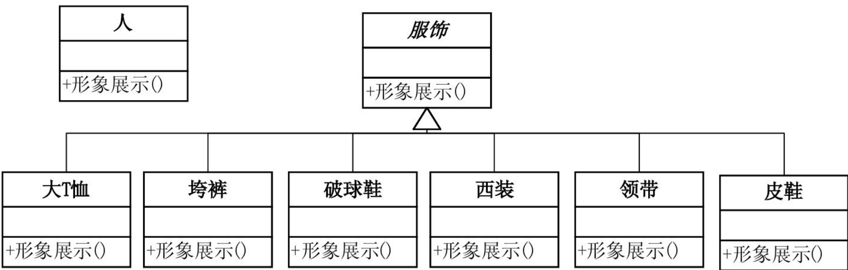
“那就改改‘Person’类就行了，”小菜说完就反应过来，“哦，不对，这就违背了开放-封闭原则了。哈，我知道了，应该把这些服饰都写成子类就好了。我去改。”

大鸟抬起手伸出食指对小菜点了点，“你呀，刚学的这么重要的原则，怎么还会忘？”

6.3 小菜扮靓第二版

过了不到十分钟，小菜的第二版代码出炉。

代码结构图



Person类

```
class Person
{
    private string name;
    public Person(string name)
    {
        this.name = name;
    }
    public void Show()
    {
        Console.WriteLine("装扮的{0}", name);
    }
}
```

服饰抽象类

```
//服饰
abstract class Finery
{
    public abstract void Show();
}
```

各种服饰子类

```
//大T恤
class TShirts : Finery
{
    public override void Show()
    {
        Console.Write("大T恤 ");
    }
}

//垮裤
class BigTrouser : Finery
{
    public override void Show()
    {
        Console.Write("垮裤 ");
    }
}
//其余类类似，省略
............
```

客户端代码

```
static void Main(string[] args)
{
    Person xc = new Person("小菜");
    Console.WriteLine("\n第一种装扮：");
    Finery dtx = new TShirts();
    Finery kk = new BigTrouser();
    Finery pqx = new Sneakers();

    dtx.Show();
    kk.Show();
    pqx.Show();
    xc.Show();

    Console.WriteLine("\n第二种装扮：");
    Finery xz = new Suit();
    Finery ld = new Tie();
```

```
        Finery px = new LeatherShoes();

        xz.Show();
        ld.Show();
        px.Show();
        xc.Show();

        Console.Read();
    }
```

结果显示同前例，略。

“这下你还能说我不面向对象吗？如果要加超人装扮，只要增加子类就可以了。”

“哼，用了继承，用了抽象类就算是用好了面向对象了吗？你现在的代码的确做到了‘服饰’类与‘人’类的分离，但其他问题还有存在的。”

“什么问题呢？”

“你仔细看看这段代码。”

```
        dtx.Show();
        kk.Show();
        pqx.Show();
        xc.Show();
```

“这样写意味着什么？”大鸟问道。

“就是把‘大T恤’、‘垮裤’、‘破球鞋’和‘装扮的小菜’一个词一个词的显示出来呀。”

“说得好，我要的就是你这句话，这样写就好比：你光着身子，当着大家的面，先穿T恤，再穿裤子，再穿鞋，仿佛在跳穿衣舞。难道你穿衣服都是在众目睽睽下穿的吗？”

“你的意思是，应该在内部组装完毕，然后再显示出来？这好像是建造者模式呀。”

“不是的，建造者模式要求建造的过程必须是稳定的，而现在我们这个例子，建造过程是不稳定的，比如完全可以内穿西装，外套T恤，再

加披风，打上领带，皮鞋外再穿上破球鞋；当然也完全可以只穿条裤衩就算完成。换句话就是说，通过服饰组合出一个有个性的人完全可以有无数种方案，并非是固定的。”

“啊，你说得对，其实先后顺序也是有讲究的，如你所说，先穿内裤后穿外裤，这叫凡人，内裤穿到外裤外面，那就是超人了。”

“哈，很会举一反三嘛，那你说该如何办呢？”

“我们需要把所需的功能按正确的顺序串联起来进行控制，这好像很难办哦。”

“不懂就学，其实也没什么稀罕的，这可以用一个非常有意思的设计模式来实现。”

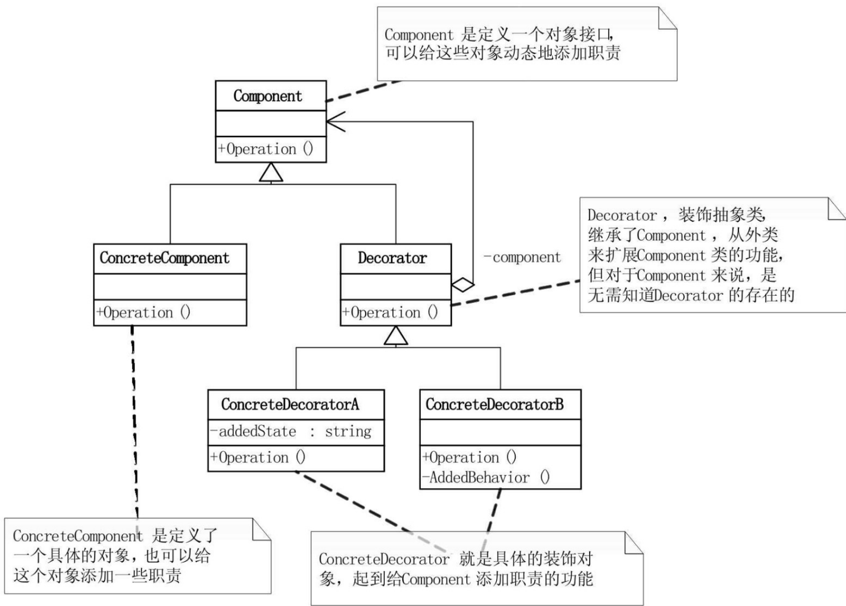
6.4 装饰模式

装饰模式（**Decorator**），动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更为灵活。[DP]

“啊，装饰这词真好，无论衣服、鞋子、领带、披风其实都可以理解为对人的装饰。”

“我们来看看它的结构。”

装饰模式（**Decorator**）结构图



“**Component**是定义一个对象接口，可以给这些对象动态地添加职责。**ConcreteComponent**是定义了一个具体的对象，也可以给这个对象添加一些职责。**Decorator**，装饰抽象类，继承了**Component**，从外类来扩展**Component**类的功能，但对于**Component**来说，是无需知道**Decorator**的存在的。至于**ConcreteDecorator**就是具体的装饰对象，起到给**Component**添加职责的功能 **[DPE]**。”

“来看看基本的代码实现。”

Component类

```
abstract class Component
{
    public abstract void Operation();
}
```

ConcreteComponent类

```
class ConcreteComponent : Component
{
    public override void Operation()
    {
        Console.WriteLine("具体对象的操作");
    }
}
```

Decorator类

```
abstract class Decorator : Component
{
    protected Component component;

    public void SetComponent(Component component)
    {
        this.component = component;
    }

    public override void Operation()
    {
        if (component != null)
        {
            component.Operation();
        }
    }
}
```

设置 Component

重写 Operation()，实际执行的是 Component 的 Operation()

ConcreteDecoratorA类

```
class ConcreteDecoratorA : Decorator
{
    private string addedState;      // 本类的独有功能，以区别于 ConcreteDecoratorB

    public override void Operation()
    {
        base.Operation();           // 首先运行原 Component 的 Operation()，再执行本类的功能，如 addedState，相当于对原 Component 进行了装饰
        addedState = "New State";
        Console.WriteLine("具体装饰对象 A 的操作");
    }
}

class ConcreteDecoratorB : Decorator
{
    public override void Operation()
    {
        base.Operation();           // 首先运行原 Component 的 Operation()，再执行本类的功能，如 AddedBehavior()，相当于对原 Component 进行了装饰
        AddedBehavior();
        Console.WriteLine("具体装饰对象 B 的操作");
    }

    private void AddedBehavior()    // 本类独有的方法，以区别于 ConcreteDecoratorA
    {

    }
}
```

客户端代码

```
static void Main(string[] args)
{
    ConcreteComponent c = new ConcreteComponent();
    ConcreteDecoratorA d1 = new ConcreteDecoratorA();
    ConcreteDecoratorB d2 = new ConcreteDecoratorB();

    d1.SetComponent(c);     // 装饰的方法是：首先用 ConcreteComponent 实例化对象 c，然后用 ConcreteDecoratorA 的实例化对象 d1 来包装 c，再用 ConcreteDecoratorB 的对象 d2 包装 d1，最终执行 d2 的 Operation()
    d2.SetComponent(d1);
    d2.Operation();

    Console.Read();
}
```

“我明白了，原来装饰模式是利用**SetComponent**来对对象进行包装的。这样每个装饰对象的实现就和如何使用这个对象分离开了，每个装饰对象只关心自己的功能，不需要关心如何被添加到对象链当中**[DPE]**。用刚才的例子来说就是，我们完全可以先穿外裤，再穿内裤，而不一定要先内后外。”

“既然你明白了，还不快些把刚才的例子改成装饰模式的代码？”大鸟提醒道。

“我还有个问题，刚才我写的那个例子中‘人’类是Component还是ConcreteComponent呢？”

“哈，学习模式要善于变通，如果只有一个**ConcreteComponent**类

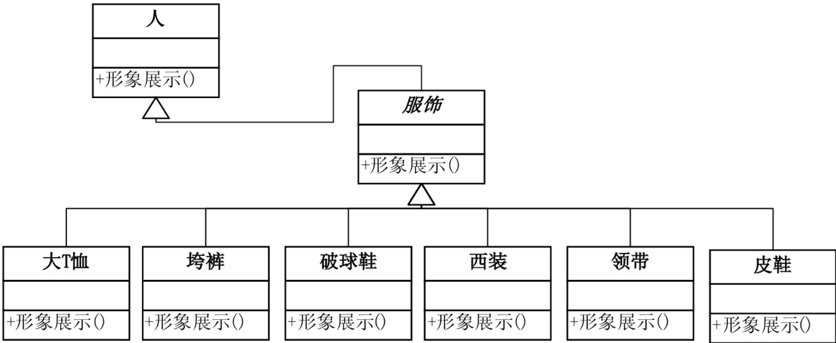
而没有抽象的**Component**类，那么**Decorator**类可以是**ConcreteComponent**的一个子类。同样道理，如果只有一个**ConcreteDecorator**类，那么就没有必要建立一个单独的**Decorator**类，而可以把**Decorator**和**ConcreteDecorator**的责任合并成一个类。”

“啊，原来如此。在这里我们就没有必要有Component类了，直接让服饰类Decorator继承人类ConcreteComponent就可。”

6.5 小菜扮靓第三版

二十分钟后，小菜第三版代码出炉。

代码结构图



"Person"类（ConcreteComponent）

```
class Person
{
    public Person()
    { }

    private string name;
    public Person(string name)
    {
        this.name = name;
    }

    public virtual void Show()
    {
        Console.WriteLine("装扮的{0}",name);
    }
}
```

服饰类（Decorator）

```
class Finery : Person
{
    protected Person component;

    //打扮
    public void Decorate(Person component)
    {
        this.component = component;
    }

    public override void Show()
    {
        if(component != null)
        {
            component.Show();
        }
    }
}
```

具体服饰类（ConcreteDecorator）

```
class TShirts : Finery
{
    public override void Show()
    {
        Console.Write("大T恤 ");
        base.Show();
    }
}

class BigTrouser : Finery
{
    public override void Show()
    {
        Console.Write("垮裤 ");
        base.Show();
    }
}

//其余类类似，省略
…………
```

客户端代码

```
static void Main(string[] args)
{
    Person xc = new Person("小菜");

    Console.WriteLine("\n 第一种装扮: ");

    Sneakers pqx = new Sneakers();
    BigTrouser kk = new BigTrouser();
    TShirts dtx = new TShirts();

    pqx.Decorate(xc);
    kk.Decorate(pqx);          装饰过程
    dtx.Decorate(kk);
    dtx.Show();

    Console.WriteLine("\n 第二种装扮: ");

    LeatherShoes px = new LeatherShoes();
    Tie ld = new Tie();
    Suit xz = new Suit();

    px.Decorate(xc);
    ld.Decorate(px);           装饰过程
    xz.Decorate(ld);
    xz.Show();

    Console.Read();
}
```

结果显示

第一种装扮：
大T恤 垮裤 破球鞋 装扮的小菜

第二种装扮：
西装 领带 皮鞋 装扮的小菜

“如果我换一种装饰方式，结果会怎样呢？”大鸟改动了小菜的代码。

```
Console.WriteLine("\n第三种装扮：");
Sneakers pqx2 = new Sneakers();
LeatherShoes px2 = new LeatherShoes();
BigTrouser kk2 = new BigTrouser();
Tie ld2 = new Tie();
```

```
pqx2.Decorate(xc);
px2.Decorate(pqx);
kk2.Decorate(px2);
ld2.Decorate(kk2);

ld2.Show();
```

结果就会显示

第三种装扮：
领带 垮裤 皮鞋 破球鞋 装扮的小菜

“哈，光着膀子、打着领带、下身垮裤、左脚皮鞋、右脚破球鞋的极具个性的小菜就展现在我们面前了。”

“你这家伙，又开始拿我开涮。我要这样子，比扮超人还要丢人。”

6.6　装饰模式总结

“来来来，总结一下，你感觉装饰模式如何？”

“我觉得装饰模式是为已有功能动态地添加更多功能的一种方式。但到底什么时候用它呢？”

“答得很好，问的问题更加好。你起初的设计中，当系统需要新功能的时候，是向旧的类中添加新的代码。这些新加的代码通常装饰了原有类的核心职责或主要行为，比如用西装或嘻哈服来装饰小菜，但这种做法的问题在于，它们在主类中加入了新的字段，新的方法和新的逻辑，从而增加了主类的复杂度，就像你起初的那个‘人’类，而这些新加入的东西仅仅是为了满足一些只在某种特定情况下才会执行的特殊行为的需要。而装饰模式却提供了一个非常好的解决方案，它把每个要装饰的功能放在单独的类中，并让这个类包装它所要装饰的对象，因此，当需要执行特殊行为时，客户代码就可以在运行时根据需要有选择地、按顺序地使用装饰功能包装对象了**[DP]**。所以就出现了上面那个例子的情况，我可以通过装饰，让你全副武装到牙齿，也可以让你只挂一丝到内裤。”

“那么装饰模式的优点我总结下来就是，把类中的装饰功能从类中搬移去除，这样可以简化原有的类。”

“是的，这样做更大的好处就是有效地把类的核心职责和装饰功能区分开了。而且可以去除相关类中重复的装饰逻辑。”

“这个模式真不错，我以后要记着常使用它。”

“你可要当心哦，装饰模式的装饰顺序很重要哦，比如加密数据和过滤词汇都可以是数据持久化前的装饰功能，但若先加密了数据再用过滤功能就会出问题了，最理想的情况，是保证装饰类之间彼此独立，这样它们就可以以任意的顺序进行组合了。”

“是呀，穿上西装再套上T恤实在不是什么好穿法。”

“明天想好了要穿什么去见MM了吗？”大鸟突然问道。

“有了装饰模式，我还用得着担心我的穿着。再说，我信奉的是《天下无贼》中刘德华说的一句经典台词：‘开好车就是好人吗？’，小菜我魅力无限，不需装饰。”

大鸟惊奇地望着小菜，无法理解地说了句：“学完模式，判若两人，你够弓虽！”

第7章　为别人做嫁衣——代理模式

7.1　为别人做嫁衣！

时间：3月17日19点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“小菜，今天见这个叫娇娇的美女见得如何呀？”大鸟一回家来就问小菜。

“嗨，别提了，人家是有男朋友的。”小菜无精打采地答道。

“有男朋友了啊，这倒是我没料到，那为什么还找你帮忙修电脑？”

“她男友叫戴励，在北京读大学呢，他们高中就开始谈恋爱了。”小菜说，“而且她还告诉了我一件比较有趣的事。”

“哦，是什么？”

“是这样的，我们在吃饭的时候，我就问她，怎么不找男友帮修电脑。她说男友在北京读书，所以没办法帮助修。我心里一想，‘你在上海怎么男友会在北京’，正想问他们是怎么认识的，她却接着问我想不想知道他男友追她的事。哈，这不正是我所希望的吗，于是我就跟着她开始了美好的回忆。”

“又不是你谈恋爱，说得这么肉麻，还‘美好的回忆’。她回忆什么了？”

“当时她是这么说的：‘那是在我高中二年级时的一天下午……’”

时间：五年前一天下午放学时　　地点：娇娇所在中学高中二年级教室　　人物：娇娇、戴励、卓贾易

“娇娇同学，这是有人送你的礼物。”一个男生手拿着一个芭比娃娃送到她的面前。

“戴励同学，这是什么意思？”娇娇望着同班的这个男生，感觉很奇怪。

“是这样的，我的好朋友，隔壁三班的卓贾易同学，请我代他送你这个礼物的。”戴励有些脸红。

“为什么要送我礼物，我不认识他呀。”

“他说……他说……他说想和你交个朋友。”戴励脸更红了，右手抓后脑勺，说话吞吞吐吐。

“不用这样，我不需要礼物的。”娇娇显然想拒绝。

“别别别，他是我最好的朋友，他请我代他送礼物给你，也是下了很大决心的，你看在我之前时常帮你辅导数学习题的面子上，就接受一下吧。”戴励有些着急。

“那好吧，今天我对解析几何的椭圆那里还是不太懂，你再给我讲讲。”娇娇提出条件后接过礼物。

“没问题，我们到教室去讲吧。”戴励松了口气。

……

几天后

“娇娇，这是卓贾易送你的花。”

……

“娇娇，这是卓贾易送你的巧克力。”

“我不要他送的东西了，我也不想和他交朋友。我愿意……我愿意和你做朋友！”娇娇终于忍不住了，直接表白。

“啊！……我不是在做梦吧……”戴励喜从天降，不敢相信。

“呆子！”娇娇微笑地骂道。

戴励用手抓了抓头发说，“其实我也喜欢你。不过……不过，那我该如何向卓贾易交待呢？”

……

从此戴励和娇娇开始恋爱了。毕业后，戴励考上了北京XX大学，而娇娇读了上海的大专。

时间：3月17日19点30分　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“喂，醒醒，还在陶醉呀。这个戴励根本就是一个大骗子，哪有什么卓贾易，这是他自己想泡MM找的借口。”大鸟不屑一顾。

“我当时也是这么想的，但她说是真的有这个人，后来那个卓贾易气死了，差点和戴励翻脸。”小菜肯定地说。

“那就不能怪戴励了，卓贾易就是为别人在做嫁衣，所以自己苦恼也是活该，谁叫他不自己主动，找人代理谈恋爱，神经病呀。”

“是呀，都怪他自己，为别人做嫁衣的滋味不好受哦。”

“这里又可以谈到一个设计模式了。”

“你不说我也知道是哪一个，代理模式对吧？”

“哈，说得没错。小菜真是越来越聪明。”

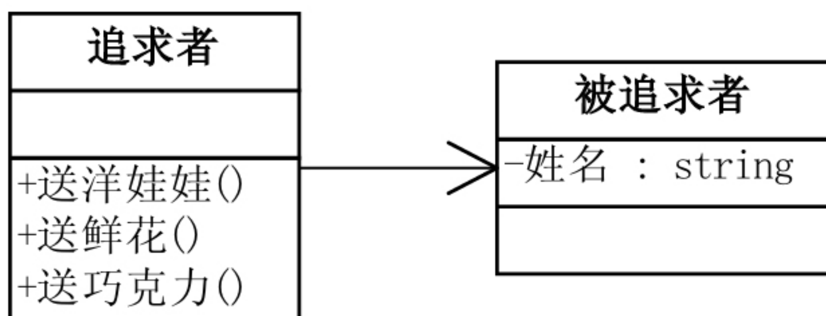
“去去去，口是心非的东西，代理模式又是怎么讲的？”

“你先试着写如果卓贾易直接追娇娇，应该如何做？”

7.2 没有代理的代码

十分钟后，小菜写出了第一份代码。

结构图



追求者类

```
class Pursuit
{
    SchoolGirl mm;
    public Pursuit(SchoolGirl mm)
    {
        this.mm = mm;
    }
    public void GiveDolls()
    {
        Console.WriteLine(mm.Name + " 送你洋娃
娃");
    }
    public void GiveFlowers()
    {
        Console.WriteLine(mm.Name + " 送你鲜花");
    }
    public void GiveChocolate()
    {
        Console.WriteLine(mm.Name + " 送你巧克
力");
    }
}
```

被追求者类

```
class SchoolGirl
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

客户端调用代码如下

```
static void Main(string[] args)
{
    SchoolGirl jiaojiao = new SchoolGirl();
    jiaojiao.Name = "李娇娇";

    Pursuit zhuojiayi = new Pursuit(jiaojiao);

    zhuojiayi.GiveDolls();
    zhuojiayi.GiveFlowers();
    zhuojiayi.GiveChocolate();

    Console.Read();
}
```


“小菜，娇娇并不认识卓贾易，这样写不就等于他们之间互相认识，并且是卓贾易亲自送东西给娇娇了吗？”

“是呀，这如何处理？”

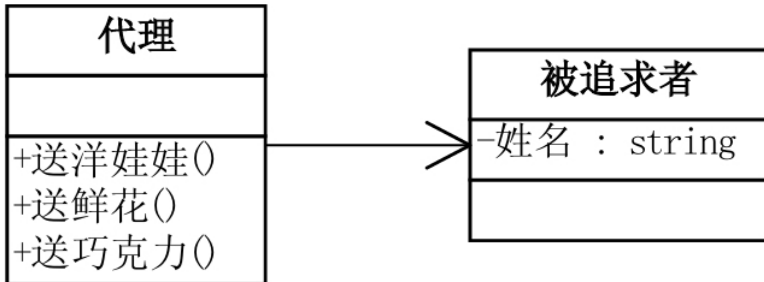
“咦，你忘了戴励了？”

“哈，对的对的，戴励就是代理呀。”

7.3 只有代理的代码

十分钟后。

结构图




```
class Proxy
{
    SchoolGirl mm;
    public Proxy(SchoolGirl mm)
    {
        this.mm = mm;
    }
    public void GiveDolls()
    {
        Console.WriteLine(mm.Name + " 送你洋娃
娃");
    }
    public void GiveFlowers()
    {
        Console.WriteLine(mm.Name + " 送你鲜花");
    }
    public void GiveChocolate()
    {
        Console.WriteLine(mm.Name + " 送你巧克
力");
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    SchoolGirl jiaojiao = new SchoolGirl();

    jiaojiao.Name = "李娇娇";

    Proxy daili = new Proxy(jiaojiao);

    daili.GiveDolls();
    daili.GiveFlowers();
    daili.GiveChocolate();

    Console.Read();
}
```

此时，“追求者”类的实例“卓贾易”又不在了

“小菜，你又犯错了。”

“这又有什么问题，为什么出错的总是我。”小菜非常不爽。

“你把‘Pursuit（追求者）’换成了‘Proxy（代理）’，把‘卓贾易’换成了‘戴励’。这就使得这个礼物变成是戴励送的，而你刚才还肯定地说，‘卓贾易’这个人是存在的，礼物是他买的，你这怎么能正确呢？”

“哦，我明白了，我这样写把‘Pursuit（追求者）’给忽略了，事实上应该是‘Pursuit（追求者）’通过‘Proxy（代理）’送给‘SchoolGirl（被追求者）’礼物，这才是合理的。那我应该如何办呢？”

“不难呀，你仔细观察一下，‘Pursuit（追求者）’和‘Proxy（代理）’有没有相似的地方？”

“他们应该都有送礼物的三个方法，只不过‘Proxy（代理）’送的礼物是‘Pursuit（追求者）’买的，实质是‘Pursuit（追求者）’送的。”

“很好，既然两者都有相同的方法，那就意味着他们都怎么样？”

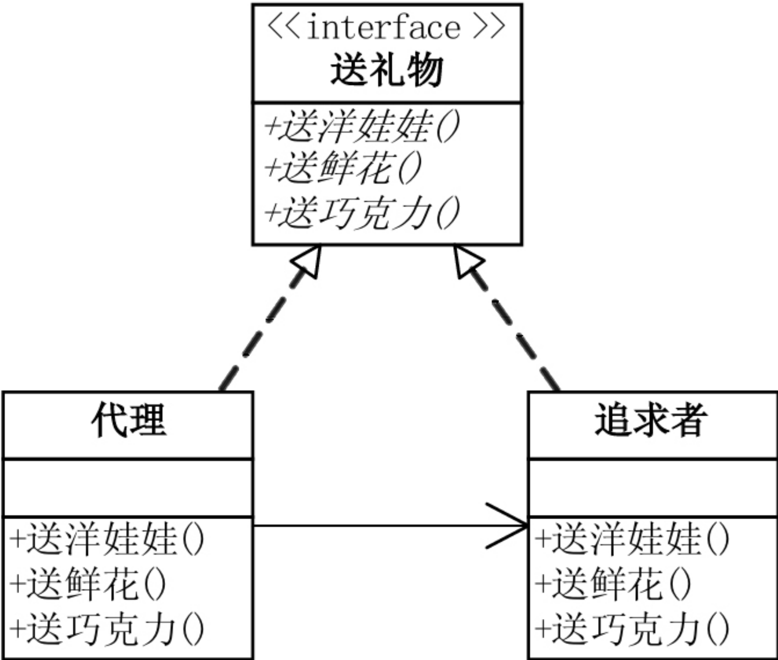
“哦，你的意思是他们都实现了同样的接口？我想，我可以写出代码来了。”

“小菜开窍了。”

7.4 符合实际的代码

十分钟后。小菜第三份代码。

结构图



代理接口如下

```
interface IGiveGift
{
    void GiveDolls();
    void GiveFlowers();
    void GiveChocolate();
}
```

追求者类如下

```
class Pursuit :IGiveGift
{
    SchoolGirl mm;
    public Pursuit(SchoolGirl mm)
    {
        this.mm = mm;
    }
    public void GiveDolls()
    {
        Console.WriteLine(mm.Name + " 送你洋娃娃");
    }
    public void GiveFlowers()
    {
        Console.WriteLine(mm.Name + " 送你鲜花");
    }
    public void GiveChocolate()
    {
        Console.WriteLine(mm.Name + " 送你巧克力");
    }
}
```

唯一变化就是让“追求者”去实现“送礼物”接口

代理类如下

```
class Proxy : IGiveGift
{
    Pursuit gg;
    public Proxy(SchoolGirl mm)
    {
        gg = new Pursuit(mm);
    }
    public void GiveDolls()
    {
        gg.GiveDolls();
    }
    public void GiveFlowers()
    {
        gg.GiveFlowers();
    }
    public void GiveChocolate()
    {
        gg.GiveChocolate();
    }
}
```

让“代理”也去实现“送礼物”接口

在实现方法中去调用“追求者”类的相关方法

客户端如下

```
static void Main(string[] args)
{
    SchoolGirl jiaojiao = new SchoolGirl();
    jiaojiao.Name = "李娇娇";

    Proxy daili = new Proxy(jiaojiao);

    daili.GiveDolls();
    daili.GiveFlowers();
    daili.GiveChocolate();

    Console.Read();
}
```

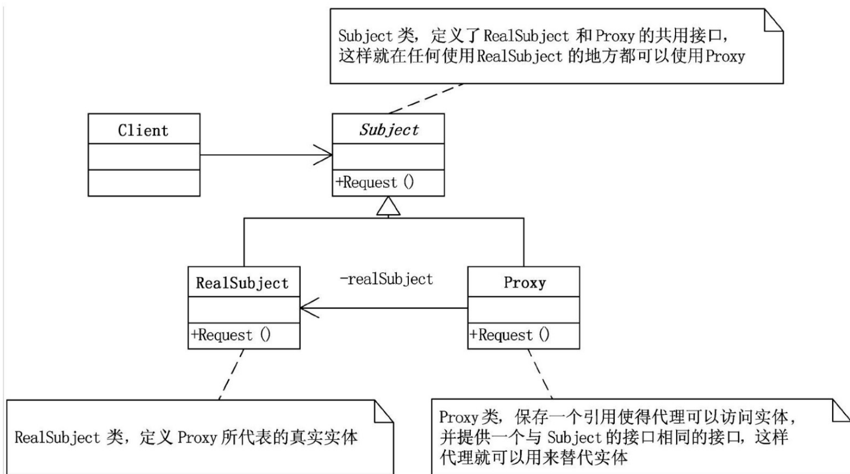
“这下好了，娇娇不认识追求她的人，但却可以通过代理人得到礼物。效果其实是达到了。”

“这就是代理模式。好了，我们来看看GoF对代理模式是如何描述的。”

7.5 代理模式

> **代理模式（Proxy）**，为其他对象提供一种代理以控制对这个对象的访问。[DP]

代理模式（**Proxy**）结构图



Subject类，定义了**RealSubject**和**Proxy**的共用接口，这样就在任何使用**RealSubject**的地方都可以使用Proxy。

```
abstract class Subject
{
    public abstract void Request();
}
```

RealSubject类，定义**Proxy**所代表的真实实体。

```
class RealSubject : Subject
{
    public override void Request()
    {
        Console.WriteLine("真实的请求");
```

```
    }
}
```

Proxy类，保存一个引用使得代理可以访问实体，并提供一个与Subject的接口相同的接口，这样代理就可以用来替代实体。

```
class Proxy : Subject
{
    RealSubject realSubject;
    public override void Request()
    {
        if(realSubject == null)
        {
            realSubject = new RealSubject();
        }
        realSubject.Request();
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    Proxy proxy = new Proxy();
    proxy.Request();

    Console.Read();
}
```

7.6　代理模式应用

“那代理模式都用在一些什么场合呢？”小菜问道。

“一般来说分为几种，第一，远程代理，也就是为一个对象在不同的地址空间提供局部代表。这样可以隐藏一个对象存在于不同地址空间的事实 **[DP]** 。”

“有没有什么例子？”

“哈，其实你是一定用过的，WebService在.NET中的应用是怎么做的？”

“哦，我明白什么叫远程代理了，当我在应用程序的项目中加入一个Web引用，引用一个WebService，此时会在项目中生成一个WebReference的文件夹和一些文件，其实它们就是代理，这就使得客户端程序调用代理就可以解决远程访问的问题。原来这就是代理模式的应用呀。”

“第二种应用是虚拟代理，是根据需要创建开销很大的对象。通过它来存放实例化需要很长时间的真实对象 **[DP]** 。这样就可以达到性能的最优化，比如说你打开一个很大的HTML网页时，里面可能有很多的文字和图片，但你还是可以很快打开它，此时你所看到的是所有的文字，但图片却是一张一张地下载后才能看到。那些未打开的图片框，就是通过虚拟代理来替代了真实的图片，此时代理存储了真实图片的路径和尺寸。”

“哦，原来浏览器当中是用代理模式来优化下载的。”

“第三种应用是安全代理，用来控制真实对象访问时的权限 **[DP]** 。一般用于对象应该有不同的访问权限的时候。第四种是智能指引，是指当调用真实的对象时，代理处理另外一些事 **[DP]** 。如计算真实对象的引用次数，这样当该对象没有引用时，可以自动释放它；或当第一次引用一个持久对象时，将它装入内存；或在访问一个实际对象前，检查是否已经锁定它，以确保其他对象不能改变它。它们都是通过代理在访问一个对象时附加一些内务处理。”

“啊，原来代理可以做这么多的事情，我还以为它是一个很不常用的模式呢。”

“代理模式其实就是在访问对象时引入一定程度的间接性，因为这种间接性，可以附加多种用途。”

“哦，明白。说白了，代理就是真实对象的代表。”

7.7 秀才让小六代其求婚

“好了，看会儿电视吧，好几天没看《武林外传》了。”大鸟打开了电视，此时武林外传正在播放第22集。

当播放到最后片段时，剧中，郭芙蓉对吕秀才恶狠狠地说：“吕秀才，是你让小六向我求婚的吧？”

“造物弄人！”吕秀才惨惨地答道，“这只是一个玩笑。”

“哦！……玩笑！”郭芙蓉冷笑地说，“我杀了你！”

秀才速奔出去，郭芙蓉口中叫着“你给我站住！”跟着跑了出去……

小菜和大鸟看到这里，转头相互看着对方，小菜说：“吕秀才让燕小六代其向郭芙蓉求婚，这不就是……”，两人异口同声的说：“代—理—模—式！”

第8章　雷锋依然在人间——工厂方法模式

8.1　再现活雷锋

时间：3月19日22点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

小菜来找大鸟，说：“今天我们见到活雷锋了。”

“哦，”大鸟感兴趣道，“现在已经很少提这个人名了，说说看。”

“我们班有个同学叫薛磊风，昨天，他出了车祸，被车撞断了腿，医生说没大碍，可以恢复。”小菜说，“四年来，他做人很低调，只听说他时常去勤工俭学，周末一般就看不到他，别的也没感觉到什么。”

“他不太走运，不过所谓‘常在路上走，哪有不碰车。’被车撞也是没办法的。”

“是呀，我们今天白天去医院看望他，却听到了一个非常感人的故事，应该说，像是在电视里才能看到的真人真事。”

“他做了感人的好事了！”

“你怎么知道，哦，我告诉你见到活雷锋的。你别打岔。”小菜不满地说，“是这样的，他这几年，都一直在帮助一个孤寡老人，每周都去老人家里，为老人洗衣扫地、买米买油，三年多，一直这样，因为他家住在广西很偏远的地方，到了南宁还要再坐几天的汽车，所以他四年来没有回过老家，也正因为此，他对这位老人家的帮助从没有停止过。”

“可是现在出了意外，他住进医院，不能帮助他人了。”大鸟还是打岔道，“这种事你骗谁呀，这么多孤寡老人，他为什么只帮助这一位？”

“你再打岔我就不说了，你不信也听听，接受一下再教育，我今天是受教育了。”小菜接着说，“他在大一的时候，从收音机里听到说，有位孤老早年丧妻，而唯一拉扯大的儿子在八十年代中越战争时牺牲了，而他正是从小生活在离战争不远的边防小镇，目睹过战争的残酷和解放军的无私奉献。所以他一听到这个消息就主动去为老人提供帮助。老人身体不好，尽管有些社会的救济，但在生活上还是不太方便，于是他帮助

老人就成了每周末的功课。”

“哦，奉献精神代代传呀。”

“现在他住院了，至少近几个月都没办法去老人家，所以他委托我们去帮他继续做这件好事，此时我们大家才知道，雷锋原来就在我们身边。”

“不容易，在21世纪，我还以为这样的人越来越少了呢，原来乐于助人的事迹还是比比皆是。”

“明天我和几名同学就去老人家，不过老人不认识我们，薛磊风也交待我们不要提任何人的名字，就像他一样说是学雷锋做好事就行了。”

“做了好事不留名，坚持不懈三年多，真让人佩服呀，受教育了。”大鸟感慨地说。

8.2 简单工厂模式实现

“好了，我来是为了请教你一个问题，这几天一直在研究工厂方法模式，但还是不太理解它和简单工厂的区别，感觉还不如简单工厂方便，为什么要用这个模式，到底这个模式的精髓在哪里？”

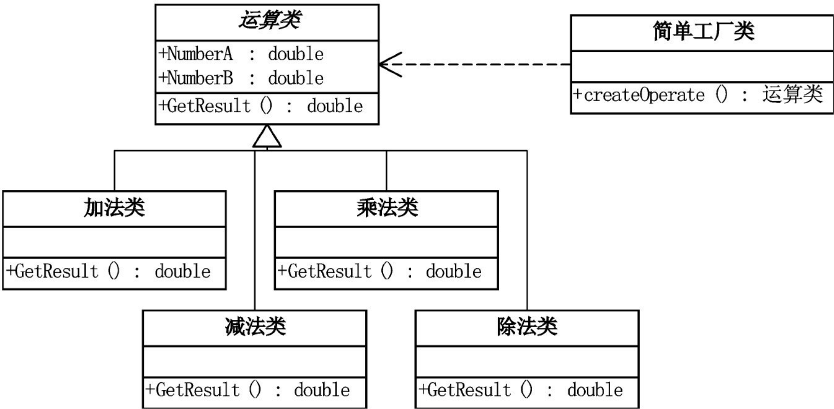
“哈，你刚才讲的故事不就是最好的工厂方法的样例吗？”

“薛磊风的事？怎么讲？”

“那你先把简单工厂模式和工厂方法模式的典型实现说给我听听。”

“哦，简单工厂模式实现是这样的。”

“首先简单工厂模式，若以我写的计算器为例，结构图如下。”



“工厂类是这样写的。”

```
class OperationFactory
{
    public static Operation createOperate(string oper
    {
        Operation oper = null;
        switch(operate)
        {
            case "+":
                oper = new OperationAdd();
                    break;
            case "-":
```

```
                oper = new OperationSub();
                break;
            case "*":
                oper = new OperationMul();
                break;
            case "/":
                oper = new OperationDiv();
                break;
        }
        return oper;
    }
}
```

“客户端的应用。”

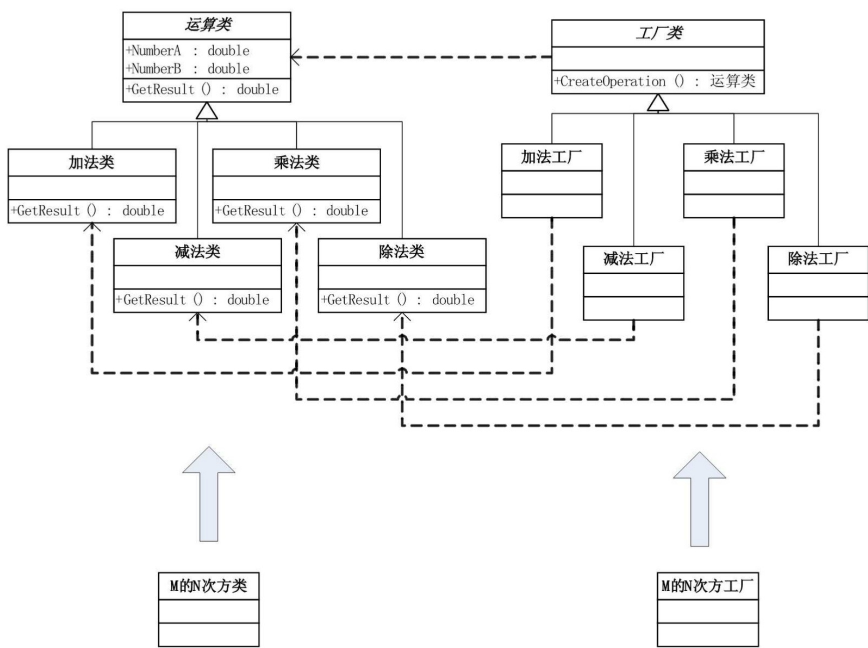
```
Operation oper;
oper = OperationFactory.createOperate("+");
oper.NumberA = 1;
oper.NumberB = 2;
double result = oper.GetResult();
```

8.3　工厂方法模式实现

“那么如果是换成工厂方法模式来写这个计算器，你能写吗？”

“当然是可以，就是因为我写出来了，才感觉好像工厂方法没什么好处！我写给你看。”

“计算器的工厂方法模式实现的结构图是这样的。”



“先构建一个工厂接口。”

```
interface IFactory
{
    Operation CreateOperation();
}
```

“然后加减乘除各建一个具体工厂去实现这个接口。”

```
class AddFactory : IFactory
{
    public Operation CreateOperation()
    {
        return new OperationAdd();
    }
}
class SubFactory : IFactory
{
    public Operation CreateOperation()
    {
        return new OperationSub();
    }
}
class MulFactory : IFactory
{
    public Operation CreateOperation()
    {
        return new OperationMul();
    }
}
class DivFactory : IFactory
{
    public Operation CreateOperation()
    {
        return new OperationDiv();
    }
}
```


“客户端的实现是这样的。”

```
IFactory operFactory = new AddFactory();
Operation oper = operFactory.CreateOperation();
oper.NumberA = 1;
oper.NumberB = 2;
double result=oper.GetResult();
```

“对呀，写得很好。”大鸟说，“工厂方法模式就是这样写的，你有什么问题？”

8.4 简单工厂vs.工厂方法

“怪就怪在这里呀，以前我们不是说过，如果我现在需要增加其他运算，比如求M数的N次方，或者求M数的N次方根，这些功能的增加，在简单工厂里，我是先去加‘求M数的N次方’功能类，然后去更改工厂方法，当中加‘Case’语句来做判断，现在用了工厂方法，加功能类没问题，再加相关的工厂类，这也没问题，但要我再去更改客户端，这不等于不但没有减化难度，反而增加了很多类和方法，把复杂性增加了吗？为什么要这样？”

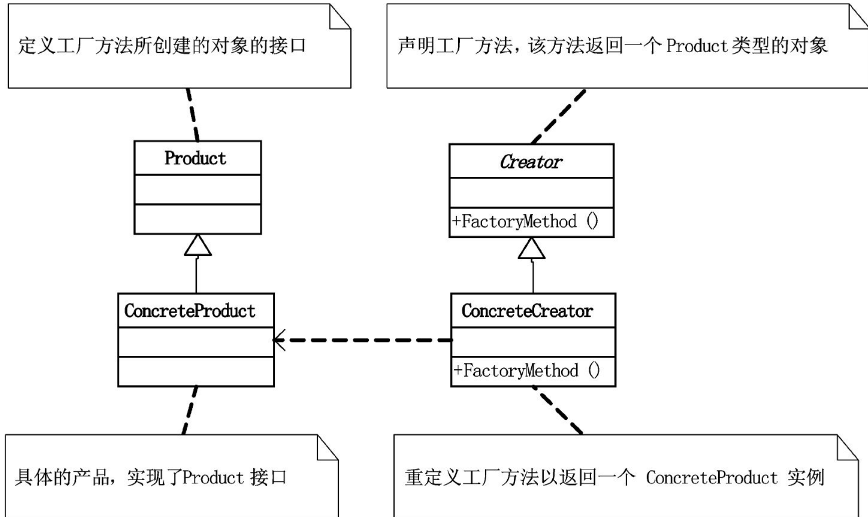
“问得好，这其实就是工厂方法模式和简单工厂的区别所在。简单工厂模式的最大优点在于工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体产品的依赖。就像你的计算器，让客户端不用管该用哪个类的实例，只需要把‘+’给工厂，工厂自动就给出了相应的实例，客户端只要去做运算就可以了，不同的实例会实现不同的运算。但问题也就在这里，如你所说，如果要加一个‘求M数的N次方’的功能，我们是一定需要给运算工厂类的方法里加‘Case’的分支条件的，修改原有的类？这可不是好办法，这就等于说，我们不但对扩展开放了，对修改也开放了，这样就违背了什么原则？”

“哦，是的，违背的是开放-封闭原则。”

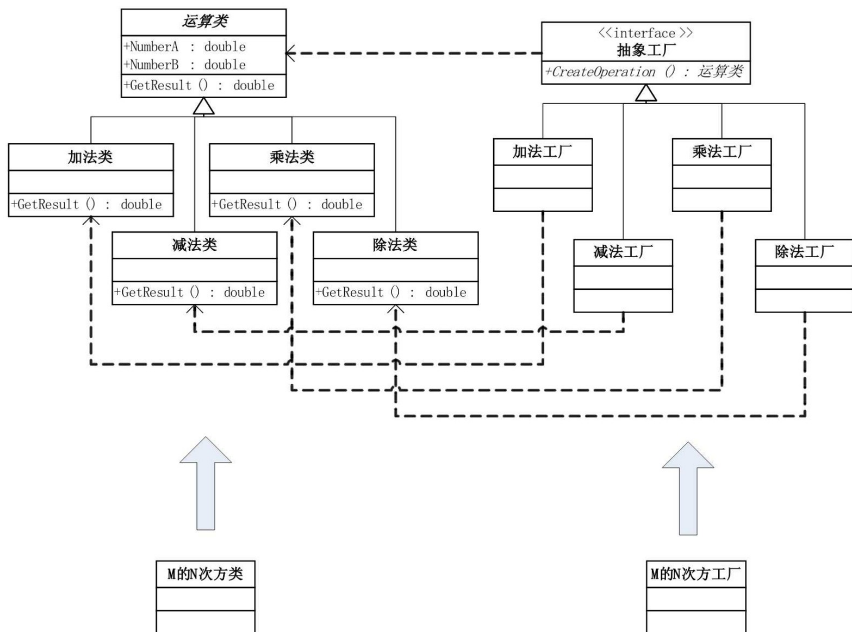
“对，于是工厂方法就来了。”

> 工厂方法模式（**Factory Method**），定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。[DP]

工厂方法模式（**Factory Method**）结构图



“我们讲过，既然这个工厂类与分支耦合，那么我就对它下手，根据依赖倒转原则，我们把工厂类抽象出一个接口，这个接口只有一个方法，就是创建抽象产品的工厂方法。然后，所有的要生产具体类的工厂，就去实现这个接口，这样，一个简单工厂模式的工厂类，变成了一个工厂抽象接口和多个具体生成对象的工厂，于是我们要增加‘求M数的N次方’的功能时，就不需要更改原有的工厂类了，只需要增加此功能的运算类和相应的工厂类就可以了。”



“这样整个工厂和产品体系其实都没有修改的变化，而只是扩展的变

化，这就完全符合了开放-封闭原则的精神。”

“哦，工厂方法从这个角度讲，的确要比简单工厂模式来得强。”

“其实你仔细观察就会发现，工厂方法模式实现时，客户端需要决定实例化哪一个工厂来实现运算类，选择判断的问题还是存在的，也就是说，工厂方法把简单工厂的内部逻辑判断移到了客户端代码来进行。你想要加功能，本来是改工厂类的，而现在是修改客户端！”

“这也是我困惑的地方。”

8.5 雷锋工厂

“这个我们过会儿再讲，以你刚才讲的故事来说吧，雷锋是众人皆知的做好人好事的模范，而你们班级的这位薛磊风同学以学习雷锋的名义做好事，而你们现在需要去代替他做好事，这其实就是典型的工厂方法模式应用了。”

“哦，说来听听。”

“首先，薛磊风作为一个大学生，以学雷锋做好事的名义去帮助老人做事，这里如何设计？”

“我的想法是这样的：

雷锋类，拥有扫地、洗衣、买米等方法。”

```
//雷锋
class LeiFeng
{
    public void Sweep()
    {
        Console.WriteLine("扫地");
    }
    public void Wash()
    {
        Console.WriteLine("洗衣");
    }
    public void BuyRice()
    {
        Console.WriteLine("买米");
    }
}
```

“‘学雷锋的大学生’类，继承‘雷锋’。”

```
//学雷锋的大学生
class Undergraduate : LeiFeng
{ }
```

“然后客户端实现的代码是这样的。”

```
LeiFeng xueleifeng = new Undergraduate();
xueleifeng.BuyRice();
xueleifeng.Sweep();
xueleifeng.Wash();
```

“小菜写得不错，”大鸟说，“现在假设你们有三个人要去代替他做这些事，那应该怎么写？”

“那就应该实例化三个‘学雷锋的大学生’对象了。”小菜说着同时写出了代码。

```
LeiFeng student1 = new Undergraduate();
student1.BuyRice();
LeiFeng student2 = new Undergraduate();
student2.Sweep();
LeiFeng student3 = new Undergraduate();
student3.Wash();
```

“你们都是要毕业的，而帮助老人却是长期工作，所以‘社区志愿者’更合适，此时这样的写法就非常不合适了，因为我们需要更改多个实例化的地方。”

“是呀，其实老人不需要知道是谁来做好事，他只需知道是学雷锋的人来帮忙就可以了。所以还需增加一个继承‘雷锋’类的‘社区志愿者’类。”

```
//社区志愿者
class Volunteer : LeiFeng
{ }
```

“再写简单工厂类。”

```
//简单雷锋工厂
```

```
class SimpleFactory
{
    public static LeiFeng CreateLeiFeng(string type)
    {
        LeiFeng result = null;
        switch(type)
        {
            case "学雷锋的大学生":
                result = new Undergraduate();
                break;
            case "社区志愿者":
                result = new Volunteer();
                break;
        }
        return result;
    }
}
```

“客户端的代码，如果要换，就只需换‘学雷锋的大学生’为‘社区志愿者’”

```
//简单工厂模式
LeiFeng studentA = SimpleFactory.CreateLeiFeng("学雷锋的大学生");
studentA.BuyRice();
LeiFeng studentB = SimpleFactory.CreateLeiFeng("学雷锋的大学生");
studentB.Sweep();
LeiFeng studentC = SimpleFactory.CreateLeiFeng("学雷锋的大学生");
studentC.Wash();
```

三句重复的代码

“好，此时你就发现了，你需要在任何实例化的时候写出这个工厂的代码。这里有重复，也就有了坏味道。你再用工厂方法模式写一遍。”

“好的，那就需要雷锋工厂了。”

```
//雷锋工厂
interface IFactory
{
    LeiFeng CreateLeiFeng();
}

//学雷锋的大学生工厂
class UndergraduateFactory : IFactory
```

```
{
    public LeiFeng CreateLeiFeng()
    {
        return new Undergraduate();
    }
}

//社区志愿者工厂
class VolunteerFactory : IFactory
{
    public LeiFeng CreateLeiFeng()
    {
        return new Volunteer();
    }
}
```

“客户端调用的时候只需要这样就可以了。”

```
//工厂方法模式
IFactory factory = new UndergraduateFactory();   // 要换成‘社区志愿者’，修改这里就可以
LeiFeng student = factory.CreateLeiFeng();

student.BuyRice();
student.Sweep();
student.Wash();
```

“我明白了，尽管如果要换成‘社区志愿者’也还是要修改代码，但是只需要修改一处就可以了。这是最佳的做法。”

“这是最佳的做法？No，No，No，不是最好的，不过应该是比较好的了。你现在明白什么时候用简单工厂模式，什么时候用工厂方法模式了吗？”

“我感觉工厂方法克服了简单工厂违背开放-封闭原则的缺点，又保持了封装对象创建过程的优点。”

“说得好，它们都是集中封装了对象的创建，使得要更换对象时，不需要做大的改动就可实现，降低了客户程序与产品对象的耦合。工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。但缺点是由于每加一个产品，就需要加一个产品工厂的类，增加了额外的开发量。”

“你说这还不是最佳的做法？那应该如何做呢？还有就是这样还是没有避免修改客户端的代码呀？”

“哈，之前我就提到过，利用‘反射’可以解决避免分支判断的问题。不过今天还是不急，等以后再谈。你明天不是还要去看望老人家吗？早点睡吧，这年头，活雷锋太少了，你们班那个薛磊风，实在是太不容易了。嗨！如果有雷锋工厂该多好。”

“雷锋工厂！雷锋工厂！是呀，如果每个人都有雷锋精神，这世界该多美好……”小菜念叨着，不觉中唱了起来，“学习雷锋，好榜样，忠于革命忠于党，爱憎分明不忘本，立场坚定斗志强……”

第9章　简历复印——原型模式

9.1　夸张的简历

时间：3月23日21点　　地点：小菜房间　　人物：小菜、大鸟

“小菜，在忙什么呢？”大鸟回家来看到小菜在整理一堆材料。

“明天要去参加一个供需见面会，所以在准备简历呢。”

“怎么这么多，可能发得出去吗？”大鸟很惊讶于小菜的简历有很厚的一叠。

“没办法呀，听其他同学说，如果简历上什么也没有，对于我们这种毕业生来说，更加不会被重视了。所以凡是能写的，我都写了，明天能多投一些就多投一些，以量取胜。另外一些准备发信件给一些报纸上登广告的企业。”

“哦，我看看。”大鸟拿起了小菜的简历，“啊，不会吧，你连小学在哪读、得了什么奖都写上去了？那干吗不把幼儿园在哪读也写上去。”

“嘿嘿！”

“C#精通、C＋＋精通、Java精通、SQL Server精通、Oracle精通，搞没搞错，你这些东西都精通？”

“其实只是学过一些，有什么办法呢，要是不写，人家就以为你什么都不懂，我写得夸张一点，可以多吸引吸引眼球吧。”

“胡闹呀，要是我是招聘的，一个稍微懂点常识的人，一看这种简历，更加不会去理会。这根本就是瞎扯嘛。”

“那你说我怎么办？我只是一个还没毕业的学生，哪来什么经验或工作经历，我能写什么？”

“哈，说得也是，对你们要求高其实也是不切实际。那你有没有准备求职信呢？”

“求职信？没考虑过，哪有空呀。再说，就写些空话、废话，只会浪费纸张。”

“你以为你现在不是在浪费纸张？你可知道，当年的我们，是如何写简历的吗？”

“不知道，难道都是手写？”

“当然，我们当年有不少同学都是手写简历和求职信，这手抄式的简历其实效果不差的，只是比较麻烦。有一次，我只写了一份简历在人才市场上转悠，身上也没带什么钱，复印就不可能了，于是在谈一家公司时，人家想留下我的简历，我却强力要求要回来，只留了个电话。”

“啊，还有你这样求职的？估计后来没戏了。”

“错，后来这家公司还真给我打电话了。回想起来，那时候对自己手写的简历很珍惜，人家公司也很重视，收到都会认真地看并答复，哪像现在。”大鸟感慨道，“印简历就像印草纸一样，发简历更像是发广告。我听说有些公司竟然在见面会结束时以拿不了为由，扔掉所收简历就走的事情，求职者要是看到岂不气晕呀。不过话说回来，像你这样自己都不重视的简历发出去，人家公司不在意也在情理之中了。”

“大鸟不会是希望我也手抄那么几十份简历吧？”

“哈，那当然没必要。毕竟时代不同了。现在程序员写简历都知道复印，在编程的时候，就不是那么多人懂得应用了。”

“哪里呀，程序员别的不一定行，Ctrl＋C到Ctrl＋V实在是太溜了，复制代码谁还不懂呀。”

“对编程来说，简单的复制粘贴极有可能造成重复代码的灾难。我所说的意思你根本还没听懂。那就以刚才的例子，我出个需求你写写看，要求有一个简历类，必须要有姓名，可以设置性别和年龄，可以设置工作经历。最终我需要写三份简历。”

“好的，我写写看。”

9.2 简历代码初步实现

二十分钟后，小菜给出了一个版本。

简历类

```
//简历
class Resume
{
    private string name;
    private string sex;
    private string age;
    private string timeArea;
    private string company;

    public Resume(string name)
    {
        this.name = name;
    }

    //设置个人信息
    public void SetPersonalInfo(string sex,
string age)
    {
        this.sex = sex;
        this.age = age;
    }

    //设置工作经历
    public void SetWorkExperience(string timeArea,
string company)
    {
        this.timeArea = timeArea;
        this.company = company;
    }

    //显示
    public void Display()
    {
        Console.WriteLine("{0} {1} {2}",name,
sex,age);
```

```
        Console.WriteLine("工作经历：{0} {1}", timeArea,company);
    }
}
```

客户端调用代码

```
static void Main(string[] args)
{
    Resume a = new Resume("大鸟");
    a.SetPersonalInfo("男","29");
    a.SetWorkExperience("1998-2000","XX公司");

    Resume b = new Resume("大鸟");
    b.SetPersonalInfo("男","29");
    b.SetWorkExperience("1998-2000","XX公司");

    Resume c = new Resume("大鸟");
    c.SetPersonalInfo("男","29");
    c.SetWorkExperience("1998-2000","XX公司");

    a.Display();
    b.Display();
    c.Display();

    Console.Read();
}
```

结果显示

```
大鸟 男 29
工作经历 1998-2000 XX公司
大鸟 男 29
工作经历 1998-2000 XX公司
大鸟 男 29
工作经历 1998-2000 XX公司
```

“很好，这其实就是当年我手写简历的时代的代码。三份简历需要三次实例化。你觉得这样的客户端代码是不是很麻烦，如果要二十份，你就需要二十次实例化。”

“是呀，而且如果我写错了一个字，比如98年改成99年，那就要改二十次。”

“你为什么不这样写呢？”

```
static void Main(string[] args)
{
    Resume a = new Resume("大鸟");
    a.SetPersonalInfo("男","29");
    a.SetWorkExperience("1998-2000","XX公司");

    Resume b = a;

    Resume c = a;

    a.Display();
    b.Display();
    c.Display();

    Console.Read();
}
```

“哈，这其实是传引用，而不是传值，这样做就如同是在b纸张和c纸张上写着简历在a处一样，没有实际的内容的。”

“不错，不错，小菜的基本功还是很扎实的。那你觉得有什么办法？”

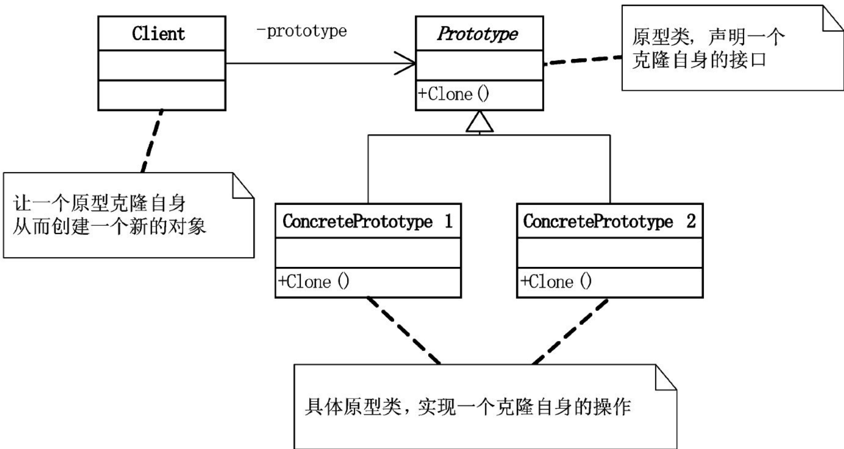
“我好像听说过有Clone克隆这样的方法，但怎么做不知道了。”

9.3 原型模式

“哈，就是它了。讲它前，要先提一个设计模式。”

原型模式（Prototype），用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。[DP]

原型模式（**Prototype**）结构图



“原型模式其实就是从一个对象再创建另外一个可定制的对象，而且不需知道任何创建的细节。我们来看看基本的原型模式代码。”

原型类

```
abstract class Prototype
{
    private string id;

    public Prototype(string id)
    {
        this.id = id;
    }

    public string Id
    {
        get { return id; }
    }

    public abstract Prototype Clone();
}
```

抽象类关键就是有这样一个 Clone 方法

具体原型类

```
class ConcretePrototype1 : Prototype
{
    public ConcretePrototype1(string id) : base(id)
    {
    }

    public override Prototype Clone()
    {
        return (Prototype)this.MemberwiseClone();
    }
}
```

创建当前对象的浅表副本。方法是创建一个新对象，然后将当前对象的非静态字段复制到该新对象。如果字段是值类型的，则对该字段执行逐位复制。如果字段是引用类型，则复制引用但不复制引用的对象；因此，原始对象及其副本引用同一对象[MSDN]

客户端代码

```
static void Main(string[] args)
{
    ConcretePrototype1 p1 = new ConcretePrototype1("I");
    ConcretePrototype1 c1 = (ConcretePrototype1)p1.Clone();
    Console.WriteLine("Cloned: {0}", c1.Id);

    Console.Read();

}
```

克隆类 ConcretePrototype1 的对象 p1 就能得到新的实例 c1

“哦，这样就可以不用实例化ConcretePrototype1了，直接克隆就行了？”小菜问道。

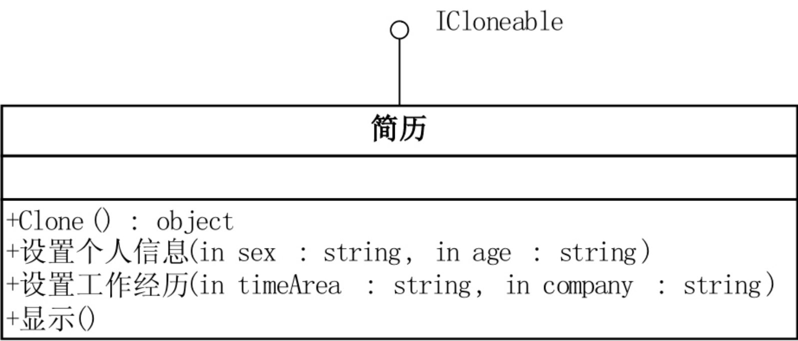
“说得没错，就是这样的。但对于.NET而言，那个原型抽象类Prototype是用不着的，因为克隆实在是太常用了，所以**.NET**在**System**命名空间中提供了**ICloneable**接口，其中就是唯一的一个方法**Clone**（），这样你就只需要实现这个接口就可以完成原型模式了。现在明白了？去改吧。”

“OK，这东西看起来不难呀。”

9.4 简历的原型实现

半小时后，小菜的第二版本代码。

代码结构图



简历类

```
//简历
class Resume : ICloneable
{
    private string name;
    private string sex;
    private string age;
    private string timeArea;
    private string company;

    public Resume(string name)
    {
        this.name = name;
    }

    //设置个人信息
    public void SetPersonalInfo(string sex, string age)
    {
        this.sex = sex;
        this.age = age;
    }
    //设置工作经历
    public void SetWorkExperience(string timeArea, string company)
```

```
    {
        this.timeArea = timeArea;
        this.company = company;
    }

    //显示
    public void Display()
    {
        Console.WriteLine("{0} {1} {2}", name, sex, age);
        Console.WriteLine("工作经历：{0} {1}", timeArea, company);
    }

    public Object Clone()
    {
        return (Object)this.MemberwiseClone();
    }
}
```

实现接口的方法，用来克隆对象

客户端调用代码

```
static void Main(string[] args)
{
    Resume a = new Resume("大鸟");
    a.SetPersonalInfo("男", "29");
    a.SetWorkExperience("1998-2000", "XX 公司");

    Resume b = (Resume)a.Clone();
    b.SetWorkExperience("1998-2006", "YY 企业");

    Resume c = (Resume)a.Clone();
    c.SetPersonalInfo("男", "24");

    a.Display();
    b.Display();
    c.Display();

    Console.Read();
}
```

只需要调用 Clone 方法就可以实现新简历的生成，并且可以再修改新简历的细节

结果显示

大鸟 男 29
工作经历 1998-2000 XX公司
大鸟 男 29
工作经历 1998-2006 YY公司
大鸟 男 24
工作经历 1998-2000 XX公司

“怎么样，大鸟，这样一来，客户端的代码就清爽很多了，而且你要是想改某份简历，只需要对这份简历做一定的修改就可以了，不会影响

到其他简历，相同的部分就不用再重复了。不过不知道这样子对性能是不是有大的提高呢？”

“当然是大大提高，你想呀，每NEW一次，都需要执行一次构造函数，如果构造函数的执行时间很长，那么多次的执行这个初始化操作就实在是太低效了。一般在初始化的信息不发生变化的情况下，克隆是最好的办法。这既隐藏了对象创建的细节，又对性能是大大的提高，何乐而不为呢？”

“的确，我开始也没感觉到它的好，听你这么一说，感觉这样做的好处还真不少，它等于是不用重新初始化对象，而是动态地获得对象运行时的状态。这个模式真的很不错。”

9.5　浅复制与深复制

“别高兴得太早，如果我现在要改需求，你就又头疼了。你现在‘简历’对象里的数据都是string型的，而string是一种拥有值类型特点的特殊引用类型，MemberwiseClone（ ）方法是这样，如果字段是值类型的，则对该字段执行逐位复制，如果字段是引用类型，则复制引用但不复制引用的对象；因此，原始对象及其复本引用同一对象。什么意思呢，就是说如果你的‘简历’类当中有对象引用，那么引用的对象数据是不会被克隆过来的。”

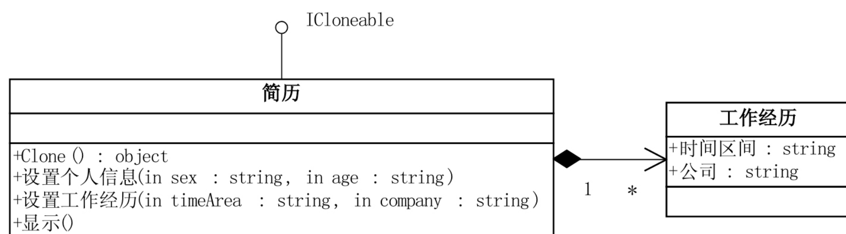
“没太听懂，为什么不能一同复制过来呢？”

“举个例子你就明白了，你现在的‘简历’类当中有一个‘设置工作经历’的方法，在现实设计当中，一般会再有一个‘工作经历’类，当中有‘时间区间’和‘公司名称’等属性，‘简历’类直接调用这个对象即可。你按照我说的再写写看。”

“好的，我试试。”

半小时后，小菜的第三个版本。

代码结构图



工作经历类

```
//工作经历
class WorkExperience
{
    private string workDate;
    public string WorkDate
    {
        get { return workDate; }
        set { workDate = value; }
    }
```

```
        private string company;
        public string Company
        {
            get { return company; }
            set { company = value; }
        }
    }
```

简历类

```
//简历
class Resume : ICloneable
{
    private string name;
    private string sex;
    private string age;

    private WorkExperience work;

    public Resume(string name)
    {
        this.name = name;
        work = new WorkExperience();
    }

    //设置个人信息
    public void SetPersonalInfo(string sex, string age)
    {
        this.sex = sex;
        this.age = age;
    }
    //设置工作经历
    public void SetWorkExperience(string workDate, string company)
    {
        work.WorkDate = workDate;
        work.Company = company;
    }

    //显示
    public void Display()
    {
        Console.WriteLine("{0} {1} {2}", name, sex, age);
        Console.WriteLine("工作经历: {0} {1}", work.WorkDate, work.Company);
    }

    public Object Clone()
    {
        return (Object)this.MemberwiseClone();
    }

}
```

引用“工作经历”对象

在“简历”类实例化时同时实例化“工作经历”

调用此方法时，给对象的两属性赋值

显示时，显示“工作经历”的两属性的值

客户端调用代码

```
static void Main(string[] args)
{
    Resume a = new Resume("大鸟");
    a.SetPersonalInfo("男", "29");
    a.SetWorkExperience("1998-2000", "XX 公司");

    Resume b = (Resume)a.Clone();
    b.SetWorkExperience("1998-2006", "YY 企业");

    Resume c = (Resume)a.Clone();
    c.SetPersonalInfo("男","24");
    c.SetWorkExperience("1998-2003", "ZZ 企业");

    a.Display();
    b.Display();
    c.Display();

    Console.Read();
}
```

b 和 c 都克隆于 a，但当它们都设置了“工作经历”时，我们希望的结果是三个的显示不一样

结果显示

```
大鸟 男 29
工作经历 1998-2003 ZZ 企业
大鸟 男 29
工作经历 1998-2003 ZZ 企业
大鸟 男 24
工作经历 1998-2003 ZZ 企业
```

可惜，没有达到我们的要求，三次显示的结果都是最后一次设置的值

“通过写代码，并且去查了一下MemberwiseClone的MSDN帮助，我大概知道你的意思了，由于它是浅表复制，所以对于值类型，没什么问题，对引用类型，就只是复制了引用，对引用的对象还是指向了原来的对象，所以就会出现我给a、b、c三个引用设置‘工作经历’，但却同时看到三个引用都是最后一次设置，因为三个引用都指向了同一个对象。”

“你写的和说的都很好，就是这个原因，这叫做‘浅复制’，被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用都仍然指向原来的对象。但我们可能更需要这样的一种需求，把要复制的对象所引用的对象都复制一遍。比如刚才的例子，我们希望是a、b、c三个引用的对象都是不同的，复制时就一变二，二变三，此时，我们就叫这种方式为‘深复制’，深复制把引用对象的变量指向复制过的新对象，而不是原有的被引用的对象。”

“那如果‘简历’对象引用了‘工作经历’，‘工作经历’再引用‘公司’，‘公司’再引用‘职位’……这样一个引用一个，很多层，如何办？”

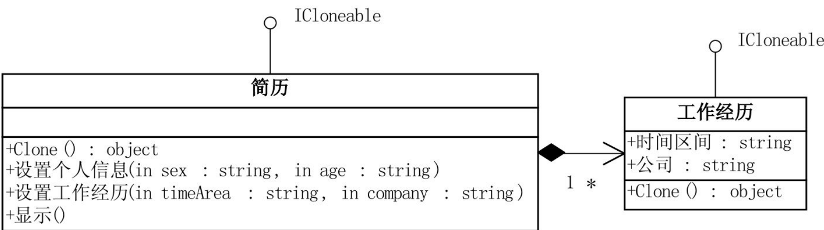
“这的确是个很难回答的问题，深复制要深入到多少层，需要事先就考虑好，而且要当心出现循环引用的问题，需要小心处理，这里比较复杂，可以慢慢研究。就现在这个例子，问题应该不大，深入到第一层就可以了。”

“那应该如何改，我没方向了。”

“好，来看我的。”

9.6 简历的深复制实现

代码结构图



工作经历类

```
//工作经历
class WorkExperience : ICloneable
{
    private string workDate;
    public string WorkDate
    {
        get { return workDate; }
        set { workDate = value; }
    }
    private string company;
    public string Company
    {
        get { return company; }
        set { company = value; }
    }

    public Object Clone()
    {
        return (Object)this.MemberwiseClone();
    }
}
```

让“工作经历”实现 ICloneable 接口

“工作经历”类实现克隆方法

简历类

```
//简历
class Resume : ICloneable
{
    private string name;
    private string sex;
    private string age;
    private WorkExperience work;

    public Resume(string name)
    {
        this.name = name;
        work = new WorkExperience();
    }
    private Resume(WorkExperience work)
    {
        this.work = (WorkExperience)work.Clone();
    }
    //设置个人信息
    public void SetPersonalInfo(string sex, string age)
    {
        this.sex = sex;
        this.age = age;
    }
    //设置工作经历
    public void SetWorkExperience(string workDate, string company)
    {
        work.WorkDate = workDate;
        work.Company = company;
    }
    //显示
    public void Display()
    {
        Console.WriteLine("{0} {1} {2}", name, sex, age);
        Console.WriteLine("工作经历: {0} {1}", work.WorkDate, work.Company);
    }
public Object Clone()
    {
        Resume obj = new Resume(this.work);
        obj.name = this.name;
        obj.sex = this.sex;
        obj.age = this.age;
        return obj;
    }
}
```

提供 Clone 方法调用的私有构造函数，以便克隆“工作经历”的数据

调用私有的构造方法，让“工作经历”克隆完成，然后再给这个“简历”对象的相关字段赋值，最终返回一个深复制的简历对象

同之前的客户端代码，其结果显示

```
大鸟 男 29
工作经历 1998-2000 XX 公司
大鸟 男 29
工作经历 1998-2006 YY 企业
大鸟 男 24
工作经历 1998-2003 ZZ 企业
```

达到了我们希望的三次显示的结果各不同的要求

“哈，原来深复制是这个意思，我明白了。”

“由于在一些特定场合，会经常涉及深复制或浅复制，比如说，数据集对象DataSet，它就有Clone（ ）方法和Copy（ ）方法，Clone（ ）方

法用来复制DataSet的结构，但不复制DataSet的数据，实现了原型模式的浅复制。Copy（ ）方法不但复制结构，也复制数据，其实就是实现了原型模式的深复制。”

9.7 复制简历vs.手写求职信

“哈，这样说来，我大量地复制我的简历，当然是原型模式的最佳体现，你的手抄时代已经结束了。”小菜得意地说。

“我倒反而认为，与其简历写得如何如何，不如认认真真地研究一下你要应聘的企业，比如看看他们的网站和对职位的要求，然后写一封比较中肯实在的求职信来得好。加上你字还写得不错，手写的求职信，更加与众不同。”

“那多累呀，也写不了多少。”

“嗨！高科技害人呀，尽管打印、复印是方便很多，所有的应聘者都这样做。但也正因为此，招聘方的重视程度也就同样低很多。如果你是手写的求职信，那就会有鹤立鸡群的效果，毕竟这样的简历或求职信太少了。”

“你说得也有道理。不过一封封地写出来感觉还是很费事呀？”

“如果是写代码，我当然会鼓励你去应用原型模式简化代码，优化设计。但对于求职，你是愿意你的简历和求职信倍受重视呢还是愿意和所有的毕业生一样千篇一律毫无新意地碰运气？”

“哈，行，听大鸟的总是没错的。那我得好好想想求职信如何写？”小菜开始拿起了笔，边写边念叨着，“亲爱的领导，冒号……”

第10章　考题抄错会做也白搭——模板方法模式

10.1　选择题不会做，蒙呗！

时间：3月27日19点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“小菜，今天面试的情况如何？”大鸟刚下班，回来就敲开了小菜的房门。

“嗨，”小菜叹了口气，“书到用时方恨少呀，英语太烂，没办法。”

“是和你用英语对话还是让你做英语题目了？”

“要是英语对话，我可能马上就跟他们说拜拜了。是做编程的英语题，因为平时英语文章看得少，所以好多单词都是似曾相识，总之猜不出意思，造成我不得不瞎蒙。还好都是选择题，一百道题蒙起来也不算太困难。”

“小菜又在指望运气了。做完后他们怎么说？”

“还不是一样，说有意向会很快与我联系。所有的公司都这样，其实一百道选择题，马上就可以算出结果来的，又何必要我多跑一趟呢。”

“题目难不难？”

“其实题目还好，如果看得懂的话，应该大多是知道的，都是些编程的基础。主要是单词记不住，所以就没把握。”

“我记得六七年前，那时候很流行微软的MCSE和MCSD的认证考试。于是国内就出现了许多的培训机构，他们弄到了微软的考试题库，给出保证通过，不通过不收费的承诺。大学生们为了能找到好工作，都去参加这个培训。我听说有个哥们，不是计算机专业的，对软件开发也算基本不懂吧，但他英文特好，于是他参加了这个培训后，短短一个多月，靠着背答案，他竟然把MCSD的证书考出来了。一个几乎不会开发的人却考出了世界最大软件公司的开发技术认证，你感觉如何？”

“说明中国学生很聪明。嘿嘿！”小菜笑道，“其实在美国，这个认证

是很有权威性的，只是中国的学生太会考试了。这带来的后果就是毁了这个证书，不管哪家公司招到这个不会开发的人都会有上当的感觉，于是对微软证书彻底失望。”

“是呀，这其实就是标准化考试的弊端。不过标准化考试好处也不少，那就是比较客观，不管世界的哪个地方，大家做同类型的题目，得分超过一定数，就判定达到一定的能力，不会因为评卷人的主观判断而影响结果。像高考的作文，由于是主观题，其实就很难说得清是好还是不好。或许不同的人给分差距是会非常大的。”

“是的，我相信鲁迅参加高考，作文一定不会得高分的。明明要是纪念，却偏要说忘却。我要是写类似的语句，一定是完了。”

“哈，大师的作品当然不能在高考这个场合去评判，高考当中写另类作文等于找死。”大鸟感慨地说，“我回想我小时候，数学老师的随堂测验，都是在黑板上抄题目，要我们先抄题目，然后再做答案，我那时候眼睛已经开始不好了，所以有时没看清楚就会把题目抄错，比如数字3我看成了8，7看成了1，那就意味着我做得再好，也不会正确了。惨呀，没考好，回家父母还说我考试成绩差是不认真学习，还专门找借口。”

“看来大鸟的往事不堪回首呀。”

“嗨，往事不要再提——你分析一下原因在哪里？”

“题目抄错了，那就不是考试题目了，而考试试卷最大的好处就是，大家都是一样的题目，特别是标准化的考试，比如全是选择或判断的题目，那就最大化地限制了答题者的发挥，大家都是ABCD或打勾打叉，非对即错的结果。”

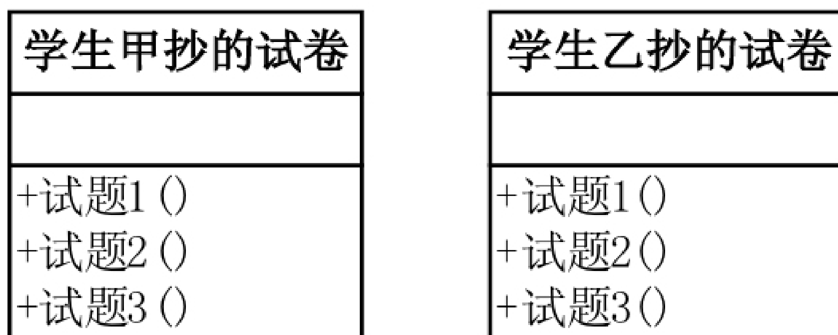
“说得好，这其实就是一个典型的设计模式。不过为了讲解这个模式，你先把抄题目的程序写给我看看。”

“好的。”

10.2 重复＝易错＋难改

二十分钟后，小菜的第一份作业。

代码结构图



学生甲抄的试卷类

```
//学生甲抄的试卷
class TestPaperA
{
    //试题1
    public void TestQuestion1()
    {
        Console.WriteLine(" 杨过得到，后来给了郭靖，
炼成倚天剑、屠龙刀的玄铁可能是[ ] a.球磨铸铁b.
        马口铁c.高速合金钢d.碳素纤维 ");
        Console.WriteLine("答案：b");
    }
    //试题2
    public void TestQuestion2()
    {
        Console.WriteLine(" 杨过、程英、陆无双铲除了
情花，造成[ ] a.使这种植物不再害人b.使一种珍
        稀物种灭绝c.破坏了那个生物圈的生态平衡d.造成该地
区沙漠化");
        Console.WriteLine("答案：a");
    }
    //试题3
```

```
    public void TestQuestion3()
    {
        Console.WriteLine(" 蓝凤凰致使华山师徒、桃谷
六仙呕吐不止，如果你是大夫，会给他们开什么药[ ]
        a.阿司匹林b.牛黄解毒片c.氟哌酸d.让他们喝大量的
生牛奶e.以上全不对");
        Console.WriteLine("答案：c");
    }
}
```

学生乙抄的试卷类

```
//学生乙抄的试卷
class TestPaperB
{
    //试题1
    public void TestQuestion1()
    {
        Console.WriteLine(" 杨过得到，后来给了郭靖，
炼成倚天剑、屠龙刀的玄铁可能是[ ] a.球磨铸铁b.
        马口铁c.高速合金钢d.碳素纤维 ");
        Console.WriteLine("答案：d");
    }
    //试题2
    public void TestQuestion2()
    {
        Console.WriteLine(" 杨过、程英、陆无双铲除了
情花，造成[ ] a.使这种植物不再害人b.使一种珍
        稀物种灭绝c.破坏了那个生物圈的生态平衡d.造成该地
区沙漠化");
        Console.WriteLine("答案：b");
    }
    //试题3
    public void TestQuestion3()
    {
        Console.WriteLine(" 蓝凤凰致使华山师徒、桃谷
六仙呕吐不止，如果你是大夫，会给他们开什么药[ ]
        a.阿司匹林b.牛黄解毒片c.氟哌酸d.让他们喝大量的
生牛奶e.以上全不对");
        Console.WriteLine("答案：a");
    }
```

```
}
```

客户端代码

```
static void Main(string[] args)
{
    Console.WriteLine("学生甲抄的试卷：");
    TestPaperA studentA = new TestPaperA();
    studentA.TestQuestion1();
    studentA.TestQuestion2();
    studentA.TestQuestion3();

    Console.WriteLine("学生乙抄的试卷：");
    TestPaperB studentB = new TestPaperB();
    studentB.TestQuestion1();
    studentB.TestQuestion2();
    studentB.TestQuestion3();

    Console.Read();
}
```

10.3 提炼代码

“大鸟，我自己都感觉到了，学生甲和学生乙两个抄试卷类非常类似，除了答案不同，没什么不一样，这样写又容易错，又难以维护。”

“说得对，如果老师突然要改题目，那两个人就都需要改代码，如果某人抄错了，那真是糟糕之极。那你说怎么办？”

“老师出一份试卷，打印多份，让学生填写答案就可以了。在这里应该就是把试题和答案分享，抽象出一个父类，让两个子类继承于它，公共的试题代码写到父类当中，就可以了。”

“好的，写写看。”

十分钟后，小菜的第二份作业。

试卷父类代码

```
//金庸小说考题试卷
class TestPaper
{
    public void TestQuestion1()
    {
        Console.WriteLine(" 杨过得到，后来给了郭靖，
炼成倚天剑、屠龙刀的玄铁可能是[ ] a.球磨铸铁b.
        马口铁c.高速合金钢d.碳素纤维 ");
    }

    public void TestQuestion2()
    {
        Console.WriteLine(" 杨过、程英、陆无双铲除了
情花，造成[ ] a.使这种植物不再害人b.使一种珍
        稀物种灭绝c.破坏了那个生物圈的生态平衡d.造成该地
区沙漠化");
    }

    public void TestQuestion3()
    {
        Console.WriteLine(" 蓝凤凰致使华山师徒、桃谷
六仙呕吐不止，如果你是大夫，会给他们开什么药[ ]
```

```
            a.阿司匹林b.牛黄解毒片c.氟哌酸d.让他们喝大量的
生牛奶e.以上全不对");
        }
    }
```

学生子类代码

```
//学生甲抄的试卷
class TestPaperA : TestPaper
{
    public new void TestQuestion1()
    {
        base.TestQuestion1();
        Console.WriteLine("答案: b");
    }

    public new void TestQuestion2()
    {
        base.TestQuestion2();
        Console.WriteLine("答案: b");
    }

    public new void TestQuestion3()
    {
        base.TestQuestion3();
        Console.WriteLine("答案: b");
    }
}
```

```
//学生乙抄的试卷
class TestPaperB : TestPaper
{
    public new void TestQuestion1()
    {
        base.TestQuestion1();
        Console.WriteLine("答案: b");
    }

    public new void TestQuestion2()
    {
        base.TestQuestion2();
        Console.WriteLine("答案: b");
    }

    public new void TestQuestion3()
    {
        base.TestQuestion3();
        Console.WriteLine("答案: b");
    }
}
```

客户端代码完全相同，略。

“大鸟，这下子类就非常简单了，只要填写答案就可以了。”

“这还只是初步的泛化，你仔细看看，两个学生的类里面，还有没有类似的代码。”

“啊，感觉相同的东西还是有的，比如都有‘base.试题1（）’，还有‘Console.WriteLine（"答案:"），我感觉除了选项的abcd，其他都是重复的。”

“说得好，我们既然用了继承，并且肯定这个继承有意义，就应该要成为子类的模板，所有重复的代码都应该要上升到父类去，而不是让每个子类都去重复。”

“那应该怎么做呢？我想不出来了。”小菜缴械投降。

“哈，模板方法登场了，当我们要完成在某一细节层次一致的一个过程或一系列步骤，但其个别步骤在更详细的层次上的实现可能不同时，我们通常考虑用模板方法模式来处理。现在来研究研究我们最初的试题方法。”

```
//试题 1
public void TestQuestion1()
{
    Console.WriteLine(" 杨过得到，后来给了郭靖，炼成倚天剑、屠龙刀的玄铁可能是[ ] a.球磨铸铁
    b.马口铁 c.高速合金钢 d.碳素纤维 ");
    Console.WriteLine("答案: b");
}
```

只有这里不同的学生会有不同的结果，其他全部都是一样的

于是我们就改动这里，增加一个虚方法。

```
public void TestQuestion1()
{
    Console.WriteLine(" 杨过得到，后来给了郭靖，炼成倚天剑、屠龙刀的玄铁可能是[ ] a.球磨铸铁
    b.马口铁 c.高速合金钢 d.碳素纤维 ");
    Console.WriteLine("答案: " + Answer1());
}

protected virtual string Answer1()
{
    return "";
}
```

改成一个虚方法

此方法的目的就是给继承的子类重写，因为这里每个人的答案都是不同的

“其余两个题目也用相同的做法。”

“然后子类就非常简单了，重写虚方法后，把答案填上，其他什么都不用管。因为父类建立了所有重复的模板。”

```
//学生甲抄的试卷
class TestPaperA : TestPaper
{
    protected override string Answer1()
    {
        return "b";
    }

    protected override string Answer2()
    {
        return "c";
    }

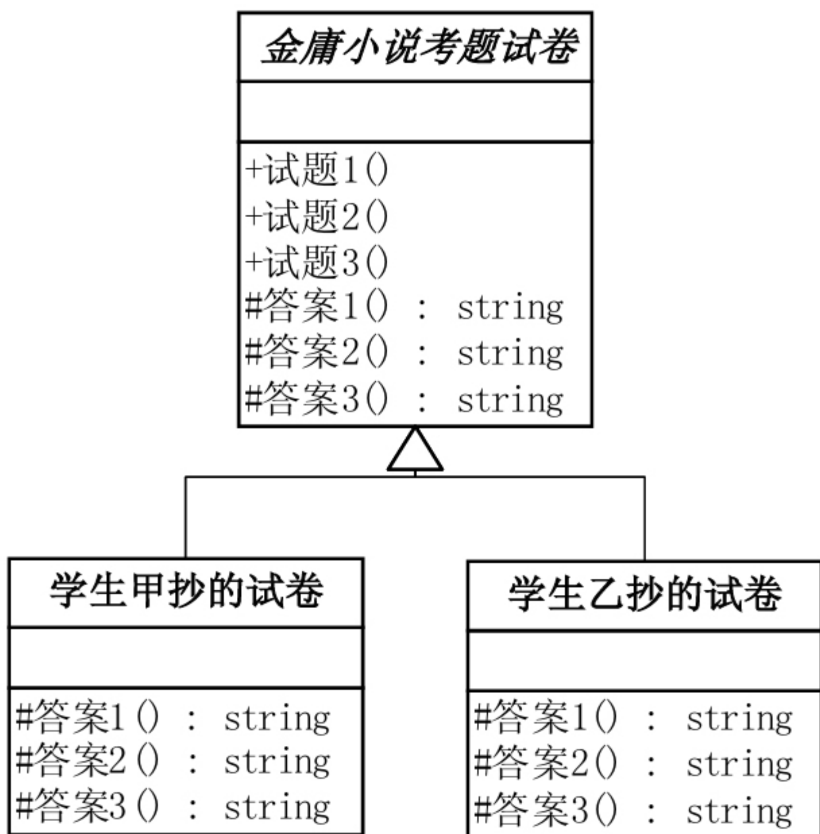
    protected override string Answer3()
    {
        return "a";
    }
}
```

```
//学生乙抄的试卷
class TestPaperB : TestPaper
{
    protected override string Answer1()
    {
        return "c";
    }

    protected override string Answer2()
    {
        return "a";
    }

    protected override string Answer3()
    {
        return "a";
    }
}
```

代码结构图



“客户端代码需要改动一个小地方，即本来是子类变量的声明，改成了父类，这样就可以利用多态性实现代码的复用了。”

```
static void Main(string[] args)
{
    Console.WriteLine("学生甲抄的试卷：");
    TestPaper studentA = new TestPaperA();
    studentA.TestQuestion1();
    studentA.TestQuestion2();
    studentA.TestQuestion3();

    Console.WriteLine("学生乙抄的试卷：");
    TestPaper studentB = new TestPaperB();
    studentB.TestQuestion1();
    studentB.TestQuestion2();
    studentB.TestQuestion3();

    Console.Read();
}
```

将子类变量的声明改成了父类，利用了多态性，实现了代码的复用

“此时要有更多的学生来答试卷，只不过是在试卷的模板上填写选择题的选项答案，这是每个人的试卷唯一的不同。”大鸟说道。

“大鸟太绝对了吧，还有姓名是不相同的吧。”

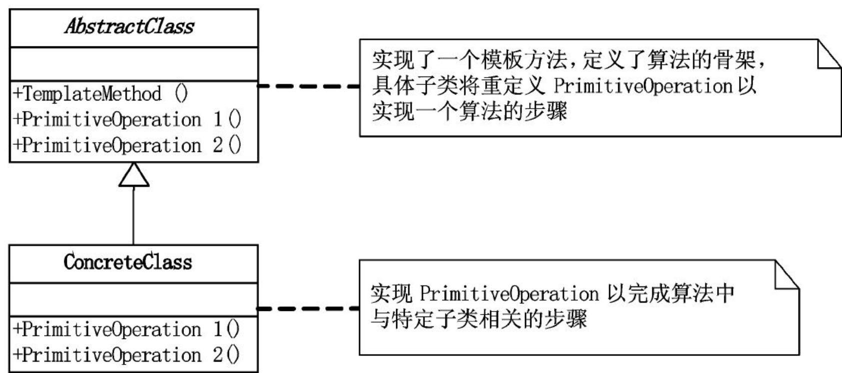
“哈，小菜说得对，除了题目答案，每个人的姓名也是不相同的。但这样的做法的的确确是对试卷的最大复用。”

10.4 模板方法模式

“而这其实就是典型的模板方法模式。”

> **模板方法模式**，定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。[DP]

模板方法模式（**TemplateMethod**）结构图



AbstractClass是抽象类，其实也就是一抽象模板，定义并实现了一个模版方法。这个模版方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

```
abstract class AbstractClass
{
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    public void TemplateMethod()
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
        Console.WriteLine("");
    }
}
```

一些抽象行为，放到子类去实现

模板方法，给出了逻辑的骨架，而逻辑的组成是一些相应的抽象操作，它们都推迟到子类实现

ConcreteClass，实现父类所定义的一个或多个抽象方法。每一个**AbstractClass**都可以有任意多个**ConcreteClass**与之对应，而每一个**ConcreteClass**都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

```
class ConcreteClassA : AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("具体类 A 方法 1 实现");
    }
    public override void PrimitiveOperation2()
    {
        Console.WriteLine("具体类 A 方法 2 实现");
    }
}

class ConcreteClassB : AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("具体类 B 方法 1 实现");
    }
    public override void PrimitiveOperation2()
    {
        Console.WriteLine("具体类 B 方法 2 实现");
    }
}
```

与 ConcreteClassB 不同的方法实现

与 ConcreteClassA 不同的方法实现

客户端调用

```
static void Main(string[] args)
{
    AbstractClass c;

    c = new ConcreteClassA();
    c.TemplateMethod();

    c = new ConcreteClassB();
    c.TemplateMethod();

    Console.Read();
}
```

10.5　模板方法模式特点

“大鸟，是不是可以这么说，模板方法模式是通过把不变行为搬移到超类，去除子类中的重复代码来体现它的优势。”

“对的，模板方法模式就是提供了一个很好的代码复用平台。因为有时候，我们会遇到由一系列步骤构成的过程需要执行。这个过程从高层次上看是相同的，但有些步骤的实现可能不同。这时候，我们通常就应该要考虑用模板方法模式了。”

“你的意思也就是说，碰到这个情况，当不变的和可变的行为在方法的子类实现中混合在一起的时候，不变的行为就会在子类中重复出现。我们通过模板方法模式把这些行为搬移到单一的地方，这样就帮助子类摆脱重复的不变行为的纠缠。”

“总结得好。看来这省心的事你总是学得最快。”

“哪里哪里，这还不是大鸟教得好呀。”小菜也不忘谦虚两句，“不过老实讲，这模板方法实在不算难，我早就用过了，只不过以前不知道这也算是一个设计模式。”

“是呀，模板方法模式是很常用的模式，对继承和多态玩得好的人几乎都会在继承体系中多多少少用到它。比如在.NET或Java类库的设计中，通常都会利用模板方法模式提取类库中的公共行为到抽象类中。”

10.6　主观题，看你怎么蒙

此时，小菜手机响了。

“请问是蔡遥先生吗？”手机那边一女士的声音。

“我是，请问您是？”小菜不认识这手机号。

“我是您今天面试的XX公司的人事经理。您今天在我们公司做的面试题，我们公司开发部非常满意，希望您能明天再到我们公司复试。”

“复试？还做选择题？”小菜有点心虚。

“哦，不是的，复试会是一些主观编程的题目，应该不是大问题的。地址您也知道，明天上午10点到吧，明天见。拜拜……嘟……嘟……”

“喂！喂！喂！”小菜喂了几声，知道对方已挂了电话，不得不放下手机，对大鸟说道，“大鸟，刚才还说选择题好，容易蒙，这下不好使了，人家要复试，还是做题，而且是主观编程题，要实实在在写代码了，不能靠猜选择题蒙了。”

“哈，看来模板方法玩不起来了。你就见招拆招吧，不就是做题吗，拿出我教你的‘伎俩’，好好表现。”

“嗯，主观题，难道我就不能蒙了？等我的好消息吧。”

第11章　无熟人难办事？——迪米特法则

11.1　第一天上班

时间：4月2日19点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“回来啦！怎么样？第一天上班感受多吧。”大鸟关心地问道。

“感受真是多哦！！！”小菜一脸的不屑。

“怎么了？受委屈了吗？说说看怎么回事。”

“委屈谈不上，就感觉公司氛围不是很好。我一大早就到他们公司，正好我的主管出去办事了。人事处的小杨让我填了表后，就带我到IT部领取电脑，她向我介绍了一个叫‘小张’的同事认识，说我跟他办领取电脑的手续就可以了。小张还蛮客气，正打算要装电脑的时候，来了个电话，叫他马上去一个客户那里处理PC故障，他说要我等等，回来帮我弄。我坐了一上午，都没有见他回来，但我发现IT部其实还有两个人，他们都在电脑前，一个忙于QQ，一个好像在看新闻。我去问人事处的小杨，可不可以请其他人帮我办理领取手续，她说她现在也在忙，让我自己去找一下IT部的小李，他或许有空。我又返回IT部办公室，请小李帮忙，小李忙着回了两个QQ后才接过我领取电脑的单子，看到上面写着‘张凡芒’负责电脑领取安装工作，于是说这个事是小张负责的，他不管，叫我还是等小张回来再说吧。我就这样又像皮球一样被踢到桌边继续等待，还好我带着一本《重构》在看，不然真要郁闷死。小张快到下班的时候才回来，开始帮我装系统，加域，设置密码等，其实也就Ghost恢复再设置一下，差不多半小时就弄好了。”小菜感叹地说道，“就这样，我这人生一个最重要的第一次就这么度过了。”

“哈哈，就业、结婚、生子，人生三大事，你这第一大事的开头是够郁闷的。”大鸟同情道，“不过现实社会就是这样的，他们又不认识你，不给你面子，也是很正常的。上班可不是上学，复杂着呢。罢了，罢了，谁叫你运气不好，你的主管在公司，事情就会好办多了。”

11.2 无熟人难办事

“不过，这家公司让你感觉不好原因在于管理上存在一些问题。”大鸟接着说，“这倒是让我想起来我们设计模式的一个原则，你的这个经历完全可以体现这个原则观点。”

“哦，是什么原则？”小菜的情绪被调动了起来，“你怎么什么事都可以和软件设计模式搭界呢？”

“大鸟我显然不是吹出来的……”大鸟洋洋得意道。

“啧啧，行了行了，大鸟你强！！！不是吹的，是天生的！快点说说，什么原则？”小菜对大鸟的吹鸟腔调颇为不满，希望快些进入正题。

“你到了公司，通过人事处小杨，认识了IT部小张，这时，你已认识了两个人。但因没人介绍你并不认识IT部小李。而既然小张小李都属于IT部，本应该都可以给你装系统配账号的，但却因小张有事，而你又不认识小李，而造成你的人生第一次大大损失，你说我分析得对吧？”

“你这都是废话，都是我告诉你的事情，哪有什么分析。”小菜失望道。

“如果你同时认识小张和小李，那么任何一人有空都可以帮你搞定了，你说对吧？”

“还是废话。”

“这就说明，你得把人际关系搞好，所谓‘无熟人难办事’，如果你在IT部‘有人’，不就万事不愁了吗？”大鸟一脸坏笑。

“大鸟，你到底想说什么？我要是有关系，对公司所有人都熟悉，还用得着你说呀。”

“小菜，瞧你急的，其实我想说的是，如果IT部有一个主管，负责分配任务，不管任何需要IT部配合的工作都让主管安排，不就没有问题了吗？”大鸟开始正经起来。

“你的意思是说，如果小杨找到的是IT部的主管，那么就算小张没空，还可以通过主管安排小李去做，是吗？”

“对头（四川方言发音）。”大鸟笑着鼓励道。

“我明白了，关键在于公司里可能没有IT主管，他们都是找到谁，就请谁去工作，如果都熟悉，有事可以协调着办，如果不熟悉，那么就会出现我碰到的情况了，有人忙死，有人闲着，而我在等待。”

“没有管理，单靠人际关系协调是很难办成事的。如果公司IT部就一个小张，那什么问题也没有，只不过效率低些。后来再来个小李，那工作是叫谁去做呢？外人又不知道他们两人谁忙谁闲的，于是抱怨、推诿、批评就随风而至。要是三个人在IT部还没有管理人员，则更加麻烦了。正所谓一个和尚挑水吃，两个和尚抬水吃，三个和尚没水吃。”

“看来哪怕两个人，也应该有管理才好。我知道你的意思了，不过这是管理问题，和设计模式有关系吗？”

“急什么，还没讲完呢？就算有IT主管，如果主管正好不在办公室怎么办呢？公司几十号人用电脑，时时刻刻都有可能出故障，电话过来找主管，人不在，难道就不解决问题了？”

“这个，看来需要规章制度，不管主管在不在，谁有空先去处理，过后汇报给主管，再来进行工作协调。”小菜也学着分析起来。

“是呀，就像有人在路上被车撞了，送到医院，难道还要问清楚有没有钱才给治疗吗，‘人命大于天’呀。同样的，在现在的高科技企业，特别是软件公司，‘电脑命大于天’，开发人员工资平均算下来每天是按数百计的，耽误一天半天，实在是公司的大损失呀——所以你想过应该怎么办没有？”

“我觉得，不管认不认识IT部的人，我只要电话或亲自找到IT部，他们都应该想办法帮我解决问题。”

“好，说得没错，那你打电话时，怎么说呢？是说‘经理在吗？……小张在吗？……’，还是‘IT部是吧，我是小菜，电脑已坏，再不修理，软件歇菜。’”

“哼，你这家伙，就会拿我开心！当然是问IT部要比问具体某个人来得更好！”

“这样子一来，不管公司任何人，找IT部就可以了，无论认不认识人，反正他们会想办法找人来解决。”

“哦，我明白了，我真的明白了。你的意思是说，IT部代表是抽象类或接口，小张小李代表是具体类，之前你在分析会修电脑不会修收音机里讲的依赖倒转原则，即面向接口编程，不要面向实现编程就是这个意思？”小菜突然有顿悟的感觉，兴奋异常。

11.3 迪米特法则

“当然，这个原则也是满足的，不过我今天想讲的是一个设计原则：‘迪米特法则（**LoD**）’也叫最少知识原则。**[J&DP]**”

> **迪米特法则**（**LoD**），如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。[J&DP]

“迪米特法则首先强调的前提是在类的结构设计上，每一个类都应当尽量降低成员的访问权限 **[J&DP]**，也就是说，一个类包装好自己的private状态，不需要让别的类知道的字段或行为就不要公开。”

“哦，是的，需要公开的字段，通常就用属性来体现了。这不是封装的思想吗？”

“当然，面向对象的设计原则和面向对象的三大特性本就不是矛盾的。迪米特法则其根本思想，是强调了类之间的松耦合。就拿你今天碰到的这件事来做例子，你第一天去公司，怎么会认识IT部的人呢，如果公司有很好的管理，那么应该是人事的小杨打个电话到IT部，告诉主管安排人给小菜你装电脑，就算开始是小张负责，他临时有急事，主管也可以再安排小李来处理。同样道理，我们在程序设计时，类之间的耦合越弱，越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成波及。也就是说，信息的隐藏促进了软件的复用。”

“明白，由于IT部是抽象的，哪怕里面的人都离职换了新人，我的电脑出问题也还是可以找IT部解决，而不需要认识其中的同事，纯靠关系帮忙了。就算需要认识，我也只要认识IT部的主管就可以了，由他来安排工作。”

“小菜动机不纯嘛！你不会是希望那个没帮你做事的小李快些被炒鱿鱼吧？哈！”大鸟瞧着小菜笑道。

“去！！！我是那样的人嘛？”小菜笑骂道。

第12章　牛市股票还会亏钱？——外观模式

12.1　牛市股票还会亏钱？

时间：4月9日19点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“大鸟，你炒股票吗？”小菜问道。

“炒过，那是好几年前了，可惜碰到熊市，亏得一塌糊涂。”大鸟坦诚地回答，“你怎么会问起股票来了？”

“我们公司的人现在都在炒股票，其实大部人都不太懂，就是因为现在股市行情很火，于是都在跟风呢！”

“那他们做得如何？”

“有一个好像还可以，赚了不少钱，具体不太清楚，但另外几个人都是刚入市的，什么都不懂，特别是一个叫顾韵梅的同事，她说得蛮搞笑的，‘今天看好了一只快涨停的股票，买进去，第二天马上就跌了。明天再去换另一只好的股票，几天都不涨，等一卖出，马上就涨停。’于是乎，在大好的牛市行情里，她却连连亏损，天天在我们面前抱怨呀。”

“哈，典型的新股民特征嘛。其实不会炒股票的话，买一只好股票放在那里所谓的‘捂股’是最好的做股票策略了。”

“自己的钱买了股票，天天都在变化，谁能不关心，特别是刚开始，都希望能涨涨涨。尽管不现实，不过赚钱的人还是有的是。不过一打开股票软件，一千多只股票，红红绿绿，又是指数大盘，又是个股K线指标，一下说基本面如何如何重要，一下又说什么有题材才可以赚大钱，头晕眼花，迷茫困惑呀。”

“小菜是不是也在做股票了？刚才提到的顾韵梅的经历，不会是说你自己的吧？”大鸟笑言道。

“哈，是真人真事，不是我。不过我也有点动心，但不知道如何炒，所以最近也下了一个软件，并小研究了一下。发现很复杂呀。”

“就算是你，也没什么不好意思的。其实股民，特别是新股民在没有足够了解证券知识的情况下去做股票，是很容易亏钱的。毕竟，需要学习的知识实在太多了，不具备这些知识就很难做好，再有就是心态也非常重要，刚开始接触股票的人一般都盼涨怕跌，于是心态很不稳定，这反而做不好股票。听说现在心理医生问病人的第一句话就是，‘你炒股票吗？’”

“要是有懂行的人帮帮忙就好了。”

“哈，基金就是你的帮手呀。它将投资者分散的资金集中起来，交由专业的经理人进行管理，投资于股票、债券、外汇等领域，而基金投资的收益归持有投资者所有，管理机构收取一定比例的托管管理费用。想想看，这样做有什么好处？”

“我感觉，由于基金会买几十支好的股票，不会因为某个股票的大跌而影响收益，尽管每个人的钱不多，但大家放在一起，反而容易达到好的投资效果。”

“说得不错，那如果是你自己做股票，为什么风险反而大了？”

“因为我需要了解股票的各种信息，需要预测它的未来，还要买入和卖出的时机合适，这其实是很难做到的。专业的基金经理人相对专业，所以就不容易像散户那么盲目。”

“尽管我们在谈股票，我还是想问问你，投资者买股票，做不好的原因和软件开发当中的什么类似？而投资者买基金，基金经理人用这些钱去做投资，然后大家获利，这其实又体现了什么？”

“我知道了，你的意思是说，由于众多投资者对众多股票的联系太多，反而不利于操作，这在软件中是不是就称为耦合性过高。而有了基金以后，变成众多用户只和基金打交道，关心基金的上涨和下跌就可以了，而实际上的操作却是基金经理人在与上千支股票和其他投资产品打交道。”

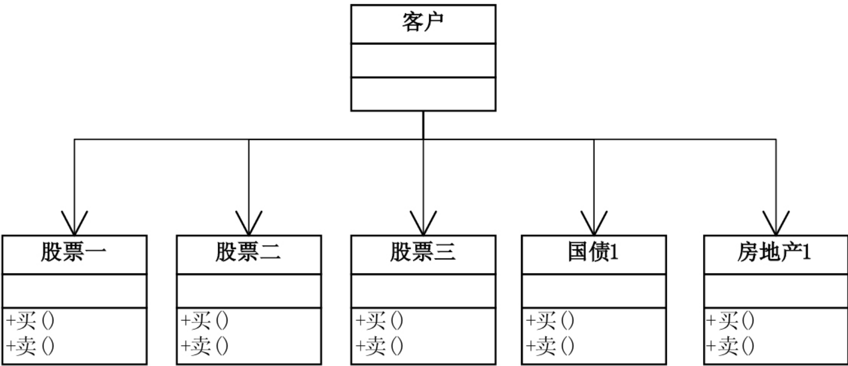
“小菜越来越不简单了嘛，这段话说得非常好，由于投资者要面对这么多的股票，又不专业，所以很难做好，但要投资者买一支好的基金，这应该是不难的，说白了，投资的目的还不是为了赚钱，那干吗不稳妥一些呢？这里其实提到了一个在面向对象开发当中用得非常多的一个设计模式——外观模式，又叫门面模式。不过为了讲清楚它，你先试着把股民炒股票的代码写写看。”

“这有何难。”

12.2　股民炒股代码

小菜写出股民投资的炒股版

代码结构图



具体股票、国债、房产类

```
//股票1
class Stock1
{
    //卖股票
    public void Sell()
    {
        Console.WriteLine(" 股票1卖出");
    }
    //买股票
    public void Buy()
    {
        Console.WriteLine(" 股票1买入");
    }
}

//股票2
class Stock2
{
    //代码类似股票1 略
}

//股票3
```

```
class Stock3
{
    //代码类似股票1 略
}

//国债1
class NationalDebt1
{
    //代码类似股票1 略
}

//房地产1
class Realty1
{
    //代码类似股票1 略
}
```

客户端调用

```
static void Main(string[] args)
{
    Stock1 gu1 = new Stock1();
    Stock2 gu2 = new Stock2();
    Stock3 gu3 = new Stock3();
    NationalDebt1 nd1 = new NationalDebt1();
    Realty1 rt1 = new Realty1();

    gu1.Buy();
    gu2.Buy();
    gu3.Buy();
    nd1.Buy();
    rt1.Buy();

    gu1.Sell();
    gu2.Sell();
    gu3.Sell();
    nd1.Sell();
    rt1.Sell();

    Console.Read();
}
```

用户需要了解股票、国债、房产情况，需要参与这些项目的具体买和卖。耦合性很高

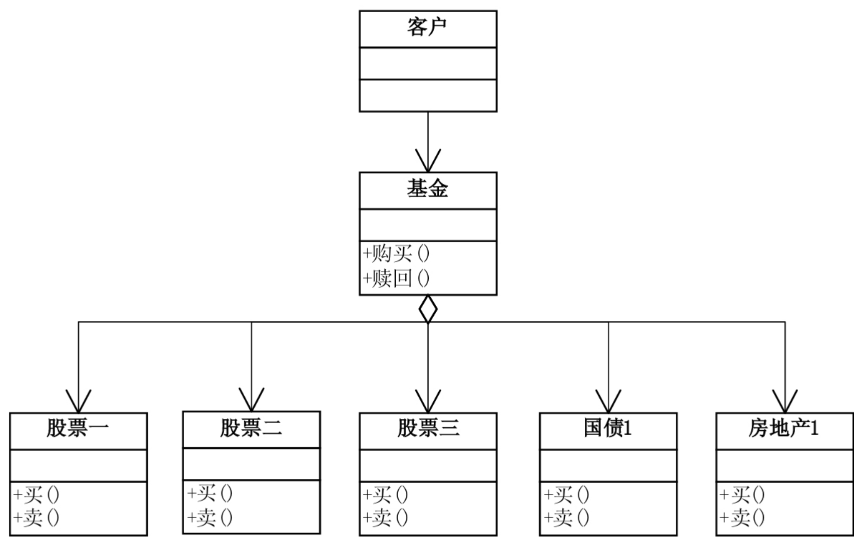
12.3 投资基金代码

“很好，如果我们现在增加基金类，将如何做？”

“那就应该是这个样子了。”

小菜写出股民投资的基金版

代码结构图



基金类如下

```
class Fund
{
    Stock1 gu1;
    Stock2 gu2;
    Stock3 gu3;
    NationalDebt1 nd1;
    Realty1 rt1;
    public Fund()
    {
        gu1 = new Stock1();
        gu2 = new Stock2();
        gu3 = new Stock3();
        nd1 = new NationalDebt1();
        rt1 = new Realty1();
    }
    public void BuyFund()
    {
        gu1.Buy();
        gu2.Buy();
        gu3.Buy();
        nd1.Buy();
        rt1.Buy();
    }
    public void SellFund()
    {
        gu1.Sell();
        gu2.Sell();
        gu3.Sell();
        nd1.Sell();
        rt1.Sell();
    }
}
```

基金类，它需要了解所有的股票或其他投资方式的方法或属性，进行组合，以备外界调用

客户端如下

```
static void Main(string[] args)
{

    Fund jijin = new Fund();
    //基金购买
    jijin.BuyFund();
    //基金赎回
    jijin.SellFund();

    Console.Read();
}
```

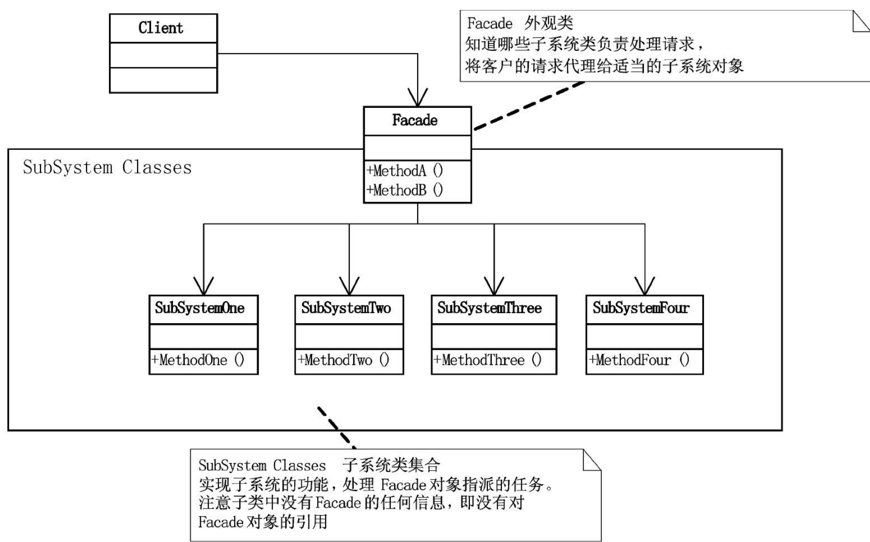
此时用户不需要了解股票，甚至可以对股票一无所知，买了基金就回家睡觉，一段时间后再赎回就可以大把数钱。参与股票的具体买卖都由基金公司完成。客户端代码非常简捷明了

“很好很好，你这样的写法，基本就是外观模式的基本代码结构了。现在我们来看看什么叫外观模式吧。”

12.4 外观模式

> 外观模式（**Facade**），为子系统中的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。[DP]

外观模式（**Facade**）结构图



四个子系统的类

```
class SubSystemOne
{
    public void MethodOne()
    {
        Console.WriteLine(" 子系统方法一");
    }
}
class SubSystemTwo
{
    public void MethodTwo()
    {
        Console.WriteLine(" 子系统方法二");
    }
}
```

```
class SubSystemThree
{
    public void MethodThree()
    {
        Console.WriteLine(" 子系统方法三");
    }
}

class SubSystemFour
{
    public void MethodFour()
    {
        Console.WriteLine(" 子系统方法四");
    }
}
```

外观类

```
class Facade
{
    SubSystemOne one;
    SubSystemTwo two;
    SubSystemThree three;
    SubSystemFour four;

    public Facade()
    {
        one = new SubSystemOne();
        two = new SubSystemTwo();
        three = new SubSystemThree();
        four = new SubSystemFour();
    }

    public void MethodA()
    {
        Console.WriteLine("\n 方法组 A() ---- ");
        one.MethodOne();
        two.MethodTwo();
        four.MethodFour();
    }

    public void MethodB()

    {
        Console.WriteLine("\n 方法组 B() ---- ");
        two.MethodTwo();
        three.MethodThree();
    }
}
```

外观类，它需要了解所有的子系统的方法或属性，进行组合，以备外界调用

客户端调用

```
static void Main(string[] args)
{
    Facade facade = new Facade();

    facade.MethodA();
    facade.MethodB();

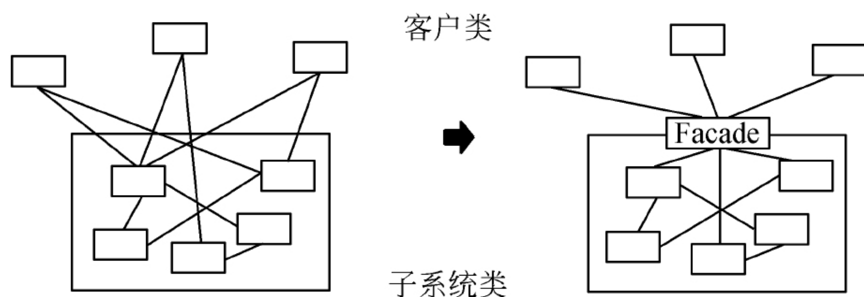
    Console.Read();
}
```


“对于面向对象有一定基础的朋友，即使没有听说过外观模式，也完全有可能在很多时候使用它，因为它完美地体现了依赖倒转原则和迪米特法则的思想，所以是非常常用的模式之一。”

12.5 何时使用外观模式

“那外观模式在什么时候使用最好呢？”小菜问道。

“这要分三个阶段来说，首先，在设计初期阶段，应该要有意识的将不同的两个层分离，比如经典的三层架构，就需要考虑在数据访问层和业务逻辑层、业务逻辑层和表示层的层与层之间建立外观**Facade**，这样可以为复杂的子系统提供一个简单的接口，使得耦合大大降低。其次，在开发阶段，子系统往往因为不断的重构演化而变得越来越复杂，大多数的模式使用时也都会产生很多很小的类，这本是好事，但也给外部调用它们的用户程序带来了使用上的困难，增加外观**Facade**可以提供一个简单的接口，减少它们之间的依赖。第三，在维护一个遗留的大型系统时，可能这个系统已经非常难以维护和扩展了，但因为它包含非常重要的功能，新的需求开发必须要依赖于它。此时用外观模式Facade也是非常合适的。你可以为新系统开发一个外观**Facade**类，来提供设计粗糙或高度复杂的遗留代码的比较清晰简单的接口，让新系统与**Facade**对象交互，**Facade**与遗留代码交互所有复杂的工作。 **[R2P]** ”



“嗯，对的，对于复杂难以维护的老系统，直接去改或去扩展都可能产生很多问题，分两个小组，一个开发Facade与老系统的交互，另一个只要了解Facade的接口，直接开发新系统调用这些接口即可，确实可以减少很多不必要的麻烦。”

“OK，我明白了，明天把股票卖了，改去买基金。”小菜感觉找到了方向。

“哈，小菜呀小菜，该不是你自己炒股票亏钱了吧？”大鸟微笑地看着小菜。

“嘿嘿，”小菜脸通红，“我当然也是参与了一点点。谁叫你不教我外观模式，害得我好容易攒了点钱却白白亏了不少。”

“啊，赖我呀？”大鸟一脸无辜。

第13章　好菜每回味不同——建造者模式

13.1　炒面没放盐

时间：4月9日22点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“小菜，讲了半天，肚子饿得厉害，走，去吃夜宵去。”大鸟摸着肚子说道。

“你请客？！”

“我教了你这么多，你也不打算报答一下，还要我请客？搞没搞错。”

“啊，说得也是，这样吧，我请客，你埋单，嘻嘻！”小菜傻笑道，“我身上没带钱。”

“你这个菜穷酸，行了，我来埋单吧。”大鸟等不及了，拿起外套就往外走。

“等等我，我把电脑关一下……”

时间：4月9日22：30　　地点：小区外大排档　　人物：小菜、大鸟

小区门口大排档前。

“老板，来两份炒饭。”大鸟对大排档的老板说。

“大鸟太小气，就请吃炒饭呀，”小菜埋怨道，“我要吃炒面。”

“再说废话，你请客了哦！”大鸟瞪了小菜一眼，接着对老板说，“那就一份炒饭一份炒面吧。”

十分钟后。

“这炒饭炒得什么玩意儿，味道不够，鸡蛋那么少，估计只放了半个。”大鸟吃得很不爽，抱怨道。

“炒面好好吃哦，真香。”小菜故意嘟囔着，把面吸得嗦嗦直响。

“让我尝尝，”大鸟强拉过小菜的盘子，吃了一口，“好像味道是不错，你小子运气好。 老板，再给我来盘炒面！”

五分钟后。

“炒面来—了—，客官，请慢用。”老板仿佛古时的小二一般来了一句。

“啊，老板，这炒面没放盐……”大鸟叫道。

……

时间：4月9日23：15　　地点：回小区的路上　　人物：小菜、大鸟

在回去的路上，大鸟感概道：“小菜，你知道为什么麦当劳、肯德基这些不过百年的洋快餐能在有千年饮食文化的中国发展得这么好吗？”

“他们比较规范吧，味道好吃，而且还不出错，不会出现像你今天这样，蛋炒饭不好吃，炒面干脆不放盐的情况。”

“你说得没错，麦当劳、肯德基的汉堡，不管在哪家店里吃，什么时间去吃，至少在中国，味道基本都是一样的。而我们国家，比如那道‘鱼香肉丝’，几乎是所有大小中餐饭店都有的一道菜，但却可以吃出上万种口味来，这是为什么？”

“厨师不一样呀，每个人做法不同的。”

“是的，因为厨师不同，他们学习厨艺方法不同，有人是科班出身，有人是师傅带徒弟，有人是照书下料，还有人是自我原创，哈，这样你说同样的菜名‘鱼香肉丝’，味道会一样吗？”

“还不只是这些，同一个厨师，不同时间烧出来同样的菜也不一样的，盐多盐少，炒的火候时间的长短，都是不一样的。”

“说得好，那你仔细想想，麦当劳、肯德基比我们很多中式快餐成功的原因是什么？”

“就感觉他们比较规范，具体原因也说不上来。”

“为什么你的炒面好吃，而我再要的炒面却没有放盐？这好吃不好吃由谁决定？”

“当然是烧菜的人，他感觉好，就是一盘好面，要是心情不好，或者粗心大意，就是一盘垃圾。”小菜肯定地说。

“哈，说得没错，今天我就吃了两盘垃圾，其实这里面最关键的就在于我们是吃得爽还是吃得难受都要依赖于厨师。你再想想我们设计模式的原则？”

“啊，你的意思是依赖倒转原则？抽象不应该依赖细节，细节应该依赖于抽象，由于我们要吃的菜都依赖于厨师这样的细节，所以我们就很被动。”

“：）好，那再想想，老麦老肯他们的产品，味道是由什么决定的？”

“我知道，那是由他们的工作流程决定的，由于他们制定了非常规范的工作流程，原料放多少，加热几分钟，都有严格规定，估计放多少盐都是用克来计量的。而这个工作流程是在所有的门店都必须要遵照执行的，所以我们吃到的东西不管在哪在什么时候味道都一样。这里我们要吃的食物都依赖工作流程。不过工作流程好像还是细节呀。”

“对，工作流程也是细节，我们去快餐店消费，我们用不用关心他们的工作流程？当然是不用，我们更关心的是是否好吃。你想如果老肯发现鸡翅烤得有些焦，他们会调整具体的工作流程中的烧烤时间，如果新加一种汉堡，做法都相同，只是配料不相同，工作流程是不变的，只是加了一种具体产品而已，这里工作流程怎么样？”

“对，这里工作流程可以是一种抽象的流程，具体放什么配料、烤多长时间等细节依赖于这个抽象。”

13.2 建造小人一

“给你出个题目，看看你能不能真正体会到流程的抽象。我的要求是你用程序画一个小人，这在游戏程序里非常常见，现在简单一点，要求是小人要有头、身体、两手、两脚就可以了。”

“废话，人还会有多手多脚呀，那不成了蜈蚣或螃蟹。这程序不难呀，我回去就写给你看。”

时间：4月9日23：30　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“大鸟，程序写出来了，我建立一支黄色的画笔，在pictureBox1上画了，简单了点，但功能实现了。”

```
Pen p = new Pen(Color.Yellow);
Graphics gThin = pictureBox1.CreateGraphics();
gThin.DrawEllipse(p, 50, 20, 30, 30);//头
gThin.DrawRectangle(p, 60, 50, 10, 50);//身体
gThin.DrawLine(p, 60, 50, 40, 100);//左手
gThin.DrawLine(p, 70, 50, 90, 100);//右手
gThin.DrawLine(p, 60, 100, 45, 150);//左脚
gThin.DrawLine(p, 70, 100, 85, 150);//右脚
```



“写得很快，那么我现在要你再画一个身体比较胖的小人呢。”

“那不难呀，我马上做好。”

```
Graphics gFat = pictureBox2.CreateGraphics();

gFat.DrawEllipse(p, 50, 20, 30, 30);
gFat.DrawEllipse(p, 45, 50, 40, 50);
gFat.DrawLine(p, 50, 50, 30, 100);
gFat.DrawLine(p, 80, 50, 100, 100);
gFat.DrawLine(p, 60, 100, 45, 150);
```



“啊，等等，我少画了一条腿。”

```
gFat.DrawLine(p,70,100,85,150);
```

“哈，这就和我们刚才去吃炒面一样，老板忘记了放盐，让本是非常美味的夜宵变得无趣。如果是让你开发一个游戏程序，里面的健全人物却少了一条腿，那怎么能行？”

“是呀，画人的时候，头身手脚是必不可少的，不管什么人物，开发时是不能少的。”

“你现在的代码全写在Form1.cs的窗体里，我要是需要在别的地方用这些画小人的程序怎么办？”

13.3 建造小人二

“嘿，你的意思是分离，这不难办，我建两个类，一个是瘦人的类，一个是胖人的类，不管谁都可以调用它了。”

```
class PersonThinBuilder        // 瘦人的类
{
    private Graphics g;
    private Pen p;

    public PersonThinBuilder(Graphics g, Pen p)    // 初始化时确定画板和颜色
    {
        this.g = g;
        this.p = p;
    }

    public void Build()    // 建造小人
    {
        g.DrawEllipse(p, 50, 20, 30, 30);
        g.DrawRectangle(p, 60, 50, 10, 50);
        g.DrawLine(p, 60, 50, 40, 100);
        g.DrawLine(p, 70, 50, 90, 100);
        g.DrawLine(p, 60, 100, 45, 150);
        g.DrawLine(p, 70, 100, 85, 150);
    }
}
```

“胖人的类也是相似的。然后我在客户端里就只需这样写就可以了。”

```
Pen p = new Pen(Color.Yellow);

Graphics gThin = pictureBox1.CreateGraphics();
PersonThinBuilder ptb = new PersonThinBuilder(gThin, p);
ptb.Build();

Graphics gFat = pictureBox2.CreateGraphics();
PersonFatBuilder pfb = new PersonFatBuilder(gFat, p);
pfb.Build();
```

“你这样写的确达到了可以复用这两个画小人程序的目的。”大鸟

说，“但炒面忘记放盐的问题依然没有解决。比如我现在需要你加一个高个的小人，你会不会因为编程不注意，又让他缺胳膊少腿呢？”

“是呀，最好的办法是规定，凡是建造小人，都必须要有头和身体，以及两手两脚。”

13.4 建造者模式

“你仔细分析会发现，这里建造小人的‘过程’是稳定的，都需要头身手脚，而具体建造的‘细节’是不同的，有胖有瘦有高有矮。但对于用户来讲，我才不管这些，我只想告诉你，我需要一个胖小人来游戏，于是你就建造一个给我就行了。如果你需要将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示的意图时，我们需要应用于一个设计模式，‘建造者（**Builder**）模式’，又叫生成器模式。建造者模式可以将一个产品的内部表象与产品的生成过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。如果我们用了建造者模式，那么用户就只需指定需要建造的类型就可以得到它们，而具体建造的过程和细节就不需知道了。”

> 建造者模式（**Builder**），将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。[DP]

“那怎么用建造者模式呢？”

“一步一步来，首先我们要画小人，都需要画什么？”

“头、身体、左手、右手、左脚、右脚。”

“对的，所以我们先定义一个抽象的建造人的类，来把这个过程给稳定住，不让任何人遗忘当中的任何一步。”

```
abstract class PersonBuilder
{
    protected Graphics g;
    protected Pen p;

    public PersonBuilder(Graphics g,Pen p)
    {
        this.g = g;
        this.p = p;
    }

    public abstract void BuildHead();
    public abstract void BuildBody();
    public abstract void BuildArmLeft();
    public abstract void BuildArmRight();
```

```
    public abstract void BuildLegLeft();
    public abstract void BuildLegRight();
}
```

“然后，我们需要建造一个瘦的小人，则让这个瘦子类去继承这个抽象类，那就必须去重写这些抽象方法了。否则编译器也不让你通过。”

```
class PersonThinBuilder : PersonBuilder
{
    public PersonThinBuilder(Graphics g,
Pen p) : base(g,p)
    { }

    public override void BuildHead()
    {
        g.DrawEllipse(p,50,20,30,30);
    }

    public override void BuildBody()
    {
        g.DrawRectangle(p,60,50,10,50);
    }

    public override void BuildArmLeft()
    {
        g.DrawLine(p,60,50,40,100);
    }

    public override void BuildArmRight()
    {
        g.DrawLine(p,70,50,90,100);
    }

    public override void BuildLegLeft()
    {
        g.DrawLine(p,60,100,45,150);
    }

    public override void BuildLegRight()
    {
        g.DrawLine(p,70,100,85,150);
```

```
        }
    }
```

“当然，胖人或高个子其实都是用类似的代码去实现这个类就可以了。”

“这样子，我在客户端要调用时，还是需要知道头身手脚这些方法呀？没有解决问题。”小菜不解地问。

“别急，我们还缺建造者模式中一个很重要的类，指挥者（Director），用它来控制建造过程，也用它来隔离用户与建造过程的关联。”

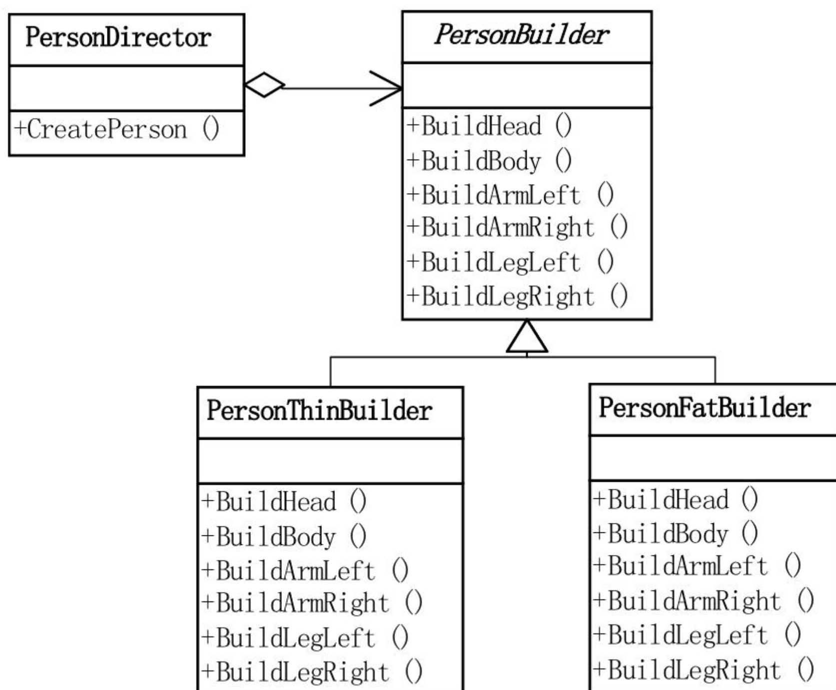
```
class PersonDirector
{
    private PersonBuilder pb;
    public PersonDirector(PersonBuilder pb)
    {
        this.pb = pb;
    }
    public void CreatePerson()
    {
        pb.BuildHead();
        pb.BuildBody();
        pb.BuildArmLeft();
        pb.BuildArmRight();
        pb.BuildLegLeft();
        pb.BuildLegRight();
    }
}
```

用户告诉指挥者，我需要什么样的小人

根据用户的选择建造小人

“你看到没有，PersonDirector类的目的就是根据用户的选择来一步一步建造小人，而建造的过程在指挥者这里完成了，用户就不需要知道了，而且，由于这个过程每一步都是一定要做的，那就不会让少画了一只手，少画一条腿的问题出现了。”

“代码结构图如下。”



“哈，我明白了，那客户端的代码我来写吧。应该也不难实现了。”

```
Pen p=new Pen(Color.Yellow);
PersonThinBuilder ptb = new PersonThinBuilder(picture
p);
PersonDirector pdThin = new PersonDirector(ptb);
pdThin.CreatePerson();

PersonFatBuilder pfb = new PersonFatBuilder(pictureBo
p);
PersonDirector pdFat = new PersonDirector(pfb);
pdFat.CreatePerson();
```

“试想一下，我如果需要增加一个高个子和矮个子的小人，我们应该怎么做？”

“加两个类，一个高个子类和一个矮个子类，让它们都去继承PersonBuilder，然后客户端调用就可以了。但我有个问题，如果我需要细化一些，比如人的五官，手的上臂、前臂和手掌，大腿小腿这些，如

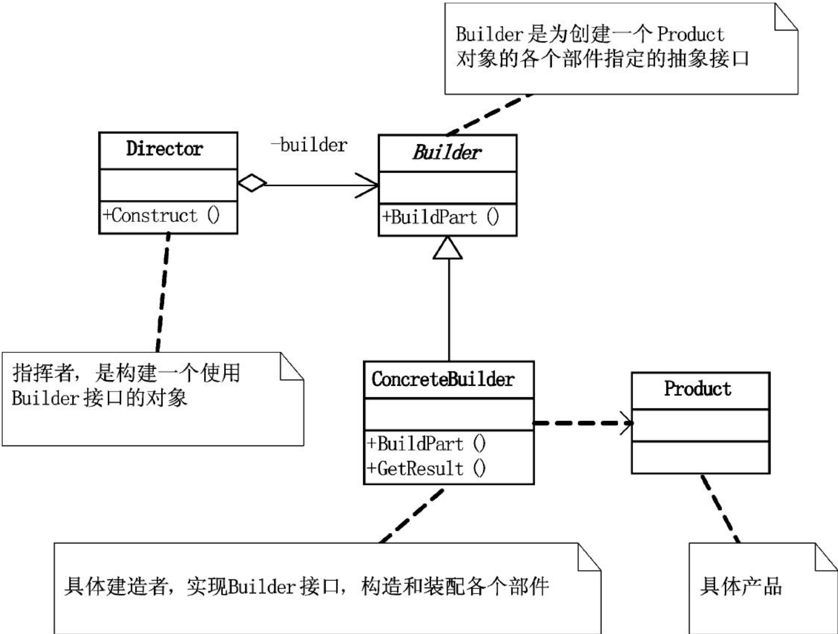
何办呢？”

“问得好，这就需要权衡，如果这些细节是每个具体的小人都需要构建的，那就应该要加进去，反之，就没必要。其实建造者模式是逐步建造产品的，所以建造者的Builder类里的那些建造方法必须要足够普遍，以便为各种类型的具体建造者构造。”

13.5 建造者模式解析

“来，我们看看建造者模式的结构。”

建造者模式（**Builder**）结构图



“现在你看这张图就不会感觉陌生了。来总结一下，**Builder**是什么？”

“是一个建造小人各个部分的抽象类。”

“概括地说，是为创建一个**Product**对象的各个部件指定的抽象接口。**ConcreteBuilder**是什么呢？”

“具体的小人建造者，具体实现如何画出小人的头身手脚各个部分。”

“对的，它是具体建造者，实现**Builder**接口，构造和装配各个部件。**Product**当然就是那些具体的小人，产品角色了，**Director**是什么？”

“指挥者，用来根据用户的需求构建小人对象。”

“嗯，它是构建一个使用**Builder**接口的对象。”

“那都是什么时候需要使用建造者模式呢？”

“它主要是用于创建一些复杂的对象，这些对象内部构建间的建造顺序通常是稳定的，但对象内部的构建通常面临着复杂的变化。”

“哦，是不是建造者模式的好处就是使得建造代码与表示代码分离，由于建造者隐藏了该产品是如何组装的，所以若需要改变一个产品的内部表示，只需要再定义一个具体的建造者就可以了。”

“来来来，我们来试着把建造者模式的基本代码推演一下，以便有一个更宏观的认识。”

13.6 建造者模式基本代码

Product类——产品类，由多个部件组成。

```
class Product
{
    IList<string> parts = new List<string>();

    public void Add(string part)        添加产品部件
    {
        parts.Add(part);
    }

    public void Show()
    {
        Console.WriteLine("\n产品 创建 ----");
        foreach (string part in parts)        列举所有的产品部件
        {
            Console.WriteLine(part);
        }
    }
}
```

Builder类——抽象建造者类，确定产品由两个部件PartA和PartB组成，并声明一个得到产品建造后结果的方法GetResult。

```
abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract Product GetResult();
}
```

ConcreteBuilder1类——具体建造者类。

```
class ConcreteBuilder1 : Builder
{
    private Product product = new Product();

    public override void BuildPartA()
    {
        product.Add("部件 A");
    }

    public override void BuildPartB()
    {
        product.Add("部件 B");
    }

    public override Product GetResult()
    {
        return product;
    }
}
```

建造具体的两个部件是部件 A 和部件 B

ConcreteBuilder2类——具体建造者类。

```
class ConcreteBuilder2 : Builder
{
    private Product product = new Product();
    public override void BuildPartA()
    {
        product.Add("部件 X");
    }

    public override void BuildPartB()
    {
        product.Add("部件 Y");
    }

    public override Product GetResult()
    {
        return product;
    }
}
```

建造具体的两个部件是部件 X 和部件 Y

Director类——指挥者类。

```
class Director
{
    public void Construct(Builder builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
    }
}
```

用来指挥建造过程

客户端代码，客户不需知道具体的建造过程。

```
static void Main(string[] args)
{
    Director director = new Director();
    Builder b1 = new ConcreteBuilder1();
    Builder b2 = new ConcreteBuilder2();

    director.Construct(b1);
    Product p1 = b1.GetResult();
    p1.Show();

    director.Construct(b2);
    Product p2 = b2.GetResult();
    p2.Show();

    Console.Read();
}
```

指挥者用 ConcreteBuilder1 的方法来建造产品

指挥者用 ConcreteBuilder2 的方法来建造产品

“所以说，建造者模式是在当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时适用的模式。”

“如果今天大排档做炒面的老板知道建造者模式，他就明白，盐是一定要放的，不然，编译就通不过。”

“什么呀，不然，钱就赚不到了，而且还大大丧失我们对他厨艺的信任。看来，各行各业都应该要懂模式呀。”

第14章　老板回来，我不知道——观察者模式

14.1　老板回来？我不知道！

时间：4月12日　21点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

小菜对大鸟说："今天白天真的笑死人了，我们一同事在上班期间看股票行情，被老板当场看到，老板很生气，后果很严重呀。"

"最近股市这么火，也应该可以理解的，你们老板说不定也炒股。"

"其实最近项目计划排得紧，是比较忙的。而最近的股市又特别的火，所以很多人都在偷偷地通过网页看行情。老板时常会出门办事，于是大家就可以轻松一些，看看行情，几个人聊聊买卖股票的心得什么的，但是一不小心，老板就会回来，让老板看到工作当中做这些总是不太好，你猜他们想到怎么办？"

"只能小心点，那能怎么办？"

"我们公司前台秘书是一个小美眉，她的名字叫童子喆，因为平时同事们买个饮料或零食什么的，都拿一份孝敬于她，所以关系比较好，现在他们就请小子喆帮忙，如果老板出门后回来，就一定要打个电话进来，大家也好马上各就各位，这样就不会被老板发现问题了。"

"哈，好主意，老板被人卧了底，这下你们那些人就不怕被发现了。"

"是呀，只要老板进门，子喆拨个电话给同事中的一个，所有人就都知道老板回来了。这种做法屡试不爽。"

"那怎么还会有今天被发现的事？"

"今天是这样的，老板出门后，大家开始个个都打开股票行情查看软件，然后还聚在一起讨论着'大盘现在如何'，'你的股票抛了没有'等事。这时老板回来后，并没有直接走进去，而是对子喆交待了几句，可能是要她打印些东西，并叫她跟老板去拿材料，这样子喆就根本没有任何时间去打电话了。"

“哈，这下完了。”

“是呀，老板带着子喆走进了办公室的时候，办公室一下子从热闹转向了安静，好几个同事本是聚在一起聊天的，赶快不说话了，回到自己的座位上，最可怜的是那个背对大门的同事——魏关姹，他显然不知道老板回来了，竟然还叫了一句‘我的股票涨停了哦。’，声音很大，就当他兴奋的转过身想表达一下激动的心情时，却看到了老板愤怒的面孔和其他同事同情的眼神。”

“幸运却又倒霉的人，谁叫他没看到老板来呢。”

“但我们老板很快恢复了笑容，平静地说道：‘魏关姹，恭喜发财呀，你是不是考虑请我们大家吃饭哦。’魏关姹面红耳赤地说，‘老板，实在对不起！以后不会了。”以后工作时还是好好工作吧。大家都继续工作吧。’老板没再说什么，就去忙事情去了。”

“啊，就这样结束了？我还当他会拿魏关姹做典型，好好批评一顿呢。不过回过头来想想看，你们老板其实很厉害，这比直接批评来得更有效，大家都是明白人，给个面子或许都能下得了台，如果真的当面批评，或许魏关姹就干不下去了。”

“是的，生气却不发作，很牛。”

14.2　双向耦合的代码

“你说的这件事的情形，是一个典型的观察者模式。”大鸟说，“你不妨把其间发生的事写成程序 看看。”

“哦，好的，我想想看。”小菜开始在纸上画起来。

半分钟后，小菜给了大鸟程序。

前台秘书类

```
class Secretary
{
    //同事列表
    private IList<StockObserver> observers = new List<StockObserver>();
    private string action;
    //增加
    public void Attach(StockObserver observer)
    {
        observers.Add(observer);
    }

    //通知
    public void Notify()
    {
        foreach (StockObserver o in observers)
            o.Update();
    }
    //前台状态
    public string SecretaryAction
    {
        get { return action; }
        set { action = value; }
    }
}
```

就是有几个同事请前台帮忙，于是就给集合增加几个对象

待老板来时，就给所有的登记的同事们发通知，“老板来了”

前台通过电话，所说的话或所做的事

看股票同事类

```
class StockObserver
{
    private string name;
    private Secretary sub;
    public StockObserver(string name, Secretary sub)
    {
        this.name = name;
        this.sub = sub;
    }
    public void Update()
    {
        Console.WriteLine("{0} {1} 关闭股票行情，继续工作！", sub.SecretaryAction, name);
    }
}
```

得到前台的通知，赶快采取行动

客户端程序如下：

```
static void Main（string[] args）
{
    //前台小姐童子喆
    Secretary tongzizhe = new Secretary（）；
    //看股票的同事
    StockObserver tongshi1 = new StockObserver（"魏关姹"，tongzizhe）；
    StockObserver tongshi2 = new StockObserver（"易管查"，tongzizhe）；

    //前台记下了两位同事
    tongzizhe.Attach（tongshi1）；
    tongzizhe.Attach（tongshi2）；
    //发现老板回来
    tongzizhe.SecretaryAction = "老板回来了！";
    //通知两个同事
    tongzizhe.Notify（）；

    Console.Read（）；
}
```

运行结果如下：

老板回来了，魏关姹 关闭股票行情，继续工作！
老板回来了，易管查 关闭股票行情，继续工作！

“写得不错，把整个事情都包括了。现在的问题是，你有没有发现，这个‘前台’类和这个‘看股票者’类之间怎么样？”

“嗯，你是不是指互相耦合？我写的时候就感觉到了，前台类要增加观察者，观察者类需要前台的状态。”

“对呀，你想想看，如果观察者当中还有人是想看NBA的网上直播（由于时差关系，美国NBA篮球比赛通常都是在北京时间的上午开始），你的‘前台’类代码怎么办？”

“那就得改动了。”

“你都发现这个问题了，你说该怎么办？想想我们的设计原则？”

“我就知道，你又要提醒我了。首先开放-封闭原则，修改原有代码就说明设计不够好。其次是依赖倒转原则，我们应该让程序都依赖抽象，而不是相互依赖。OK，我去改改，应该不难的。”

14.3 解耦实践一

半小时后，小菜给出了第二版。

增加了抽象的观察者

```
abstract class Observer
{
    protected string name;
    protected Secretary sub;

    public Observer(string name,Secretary sub)
    {
        this.name = name;
        this.sub = sub;
    }

    public abstract void Update();
}
```

增加了两个具体观察者

```
//看股票的同事
class StockObserver : Observer
{
    public StockObserver(string name,
Secretary sub) : base(name,sub)
    {
    }

    public override void Update()
    {
        Console.WriteLine("{0} {1} 关闭股票行情，继
续工作！",sub.SecretaryAction,name);
    }
}

//看NBA的同事
```

```
class NBAObserver : Observer
{
    public NBAObserver(string name,
Secretary sub) : base(name,sub)
    {
    }

    public override void Update()
    {
        Console.WriteLine("{0} {1} 关闭NBA直播，继
续工作！",sub.SecretaryAction,name);
    }
}
```

“这里让两个观察者去继承‘抽象观察者’，对于‘Update（更新）’的方法做重写操作。”

“下面是前台秘书类的编写，把所有的与具体观察者耦合的地方都改成了‘抽象观察者’。”

```
//前台秘书类
class Secretary
{
    //同事列表
    private IList<Observer> observers = new List<Observer>();
    private string action;

    //增加
    public void Attach(Observer observer)
    {
        observers.Add(observer);
    }

    //减少
    public void Detach(Observer observer)
    {
        observers.Remove(observer);
    }

    //通知
    public void Notify()
    {
        foreach (Observer o in observers)
            o.Update();
    }

    //前台状态
    public string SecretaryAction
    {
        get { return action; }
        set { action = value; }
    }
}
```

针对抽象编程，减少了与具体类的耦合

针对抽象编程，减少了与具体类的耦合

客户端代码同前面一样。

“小菜，你这样写只完成一半呀。”

“为什么，我不是已经增加了一个‘抽象观察者’了吗？”

“你小子，考虑问题为什么就不能全面点呢，你仔细看看，在具体观察者中，有没有与具体的类耦合的？”

“嗯？这里有什么？哦！我明白了，你的意思是‘前台秘书’是一个具体的类，也应该抽象出来。”

“对呀，你想想看，你们公司最后一次，你们的老板回来，前台来不及电话了，于是通知大家的任务变成谁来做？”

“是老板，对的，其实老板也好，前台也好，都是具体的通知者，这里观察者也不应该依赖具体的实现，而是一个抽象的通知者。”

“另外，就算是你们的前台，如果某一个同事和她有矛盾，她生气了，于是不再通知这位同事，此时，她是否应该把这个对象从她加入的观察者列表中删除？”

“这个容易，调用‘Detach’方法将其减去就可以了。”

“好的，再去写写看。”

14.4 解耦实践二

又过了半小时后，小菜给出了第三版。

增加了抽象通知者接口

```
//通知者接口
interface Subject
{
    void Attach(Observer observer);
    void Detach(Observer observer);
    void Notify();
    string SubjectState
    {
        get;
        set;
    }
}
```

具体的通知者类可能是前台，也可能是老板，它们也许有各自的一些方法，但对于通知者来说，它们是一样的，所以它们都去实现这个接口。

```
class Boss : Subject
{
    //同事列表
    private IList<Observer> observers = new List<Obse
    private string action;

    //增加
    public void Attach(Observer observer)
    {
        observers.Add(observer);
    }

    //减少
    public void Detach(Observer observer)
    {
```

```
        observers.Remove(observer);
    }

    //通知
    public void Notify()
    {
        foreach(Observer o in observers)
            o.Update();
    }
    //老板状态
    public string SubjectState
    {
        get { return action; }
        set { action = value; }
    }
}
```

前台秘书类与老板类类似，略。

对于具体的观察者，需更改的地方就是把与‘前台’耦合的地方都改成针对抽象通知者。

```
//抽象观察者
abstract class Observer
{
    protected string name;
    protected Subject sub;

    public Observer(string name, Subject sub)
    {
        this.name = name;
        this.sub = sub;
    }

    public abstract void Update();
}
```

原来是‘前台’，现改成‘抽象通知者’

```
//看股票的同事
class StockObserver : Observer
{
    public StockObserver(string name, Subject sub)
        : base(name, sub)
    {
    }

    public override void Update()
    {
        Console.WriteLine("{0} {1} 关闭股票行情，继续工作！", sub.SubjectState, name);
    }
}
```

原来是‘前台’，现改成‘抽象通知者’

原来是‘前台状态’，现改成‘抽象通知者状态’

客户端代码如下：

```
//老板胡汉三
Boss huhansan = new Boss();

//看股票的同事
StockObserver tongshi1 = new StockObserver("魏关姹", huhansan);
//看 NBA 的同事
NBAObserver tongshi2 = new NBAObserver("易管查", huhansan);

huhansan.Attach(tongshi1);
huhansan.Attach(tongshi2);

huhansan.Detach(tongshi1);

//老板回来
huhansan.SubjectState = "我胡汉三回来了！";
//发出通知
huhansan.Notify();
```

魏关姹其实是没有被老板通知到，所以减去

运行结果

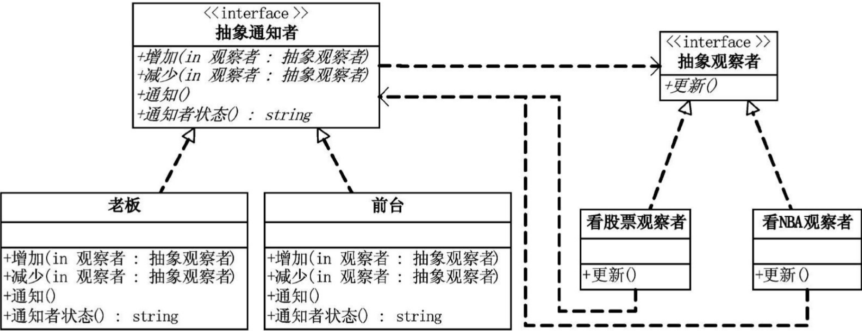
我胡汉三回来了，易管查 关闭NBA直播，继续工作！

“由于‘魏关姹’没有被通知到，所以他被当场‘抓获’，下场很惨。”小菜说道，“现在我做到了两者都不耦合了。”

“写得好，把结构图画出来看看。”

“这不难。”

小菜画出了代码的结构图。



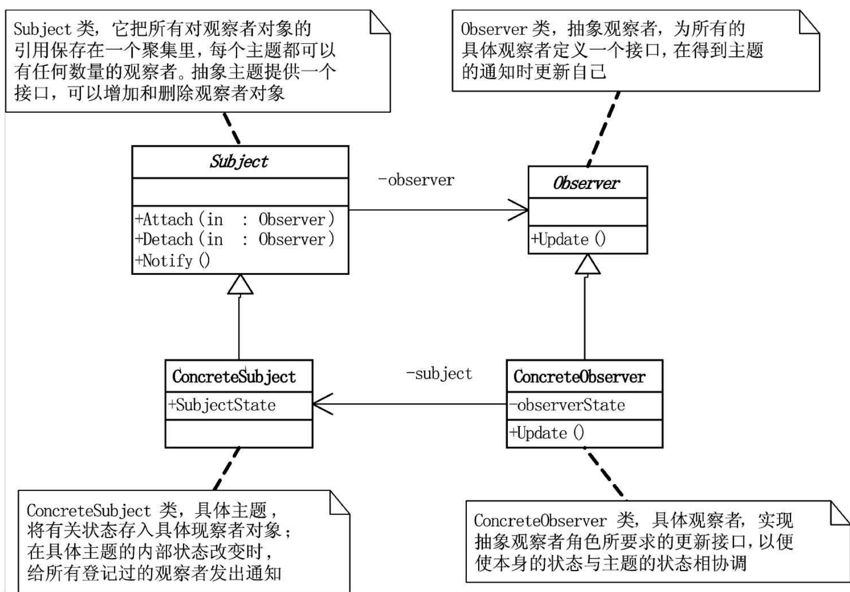
"哈，小菜非常好，你已经把观察者模式的精华都写出来了，现在我们来看看什么叫观察者模式。"

14.5 观察者模式

观察者模式又叫做发布-订阅（Publish/Subscribe）模式。

> **观察者模式**定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。[DP]

观察者模式（**Observer**）结构图



Subject类，可翻译为主题或抽象通知者，一般用一个抽象类或者一个接口实现。它把所有对观察者对象的引用保存在一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。

```
abstract class Subject
{
    private IList<Observer> observers = new List<Obse

    //增加观察者
    public void Attach(Observer observer)
    {
```

```
            observers.Add(observer);
        }
        //移除观察者
        public void Detach(Observer observer)
        {
            observers.Remove(observer);
        }
        //通知
        public void Notify()
        {
            foreach(Observer o in observers)
            {
                o.Update();
            }
        }
    }
```

Observer类，抽象观察者，为所有的具体观察者定义一个接口，在得到主题的通知时更新自己。这个接口叫做更新接口。抽象观察者一般用一个抽象类或者一个接口实现。更新接口通常包含一个Update（）方法，这个方法叫做更新方法。

```
    abstract class Observer
    {
        public abstract void Update();
    }
```

ConcreteSubject类，叫做具体主题或具体通知者，将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色通常用一个具体子类实现。

```
    class ConcreteSubject : Subject
    {
        private string subjectState;
            //具体被观察者状态
        public string SubjectState
        {
            get { return subjectState; }
```

```
            set { subjectState = value; }
        }
    }
```

ConcreteObserver类，具体观察者，实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。具体观察者角色可以保存一个指向具体主题对象的引用。具体观察者角色通常用一个具体子类实现。

```
    class ConcreteObserver : Observer
    {
        private string name;
        private string observerState;
        private ConcreteSubject subject;

        public ConcreteObserver(ConcreteSubject subject,
    string name)
        {
            this.subject = subject;
            this.name = name;
        }

        public override void Update()
        {
            observerState = subject.SubjectState;
            Console.WriteLine("观察者{0}的新状态是
    {1}",name,observerState);
        }

        public ConcreteSubject Subject
        {
            get { return subject; }
            set { subject = value; }
        }
    }
```

客户端代码

```
static void Main(string[] args)
{
    ConcreteSubject s = new ConcreteSubject();

    s.Attach(new ConcreteObserver(s,"X"));
    s.Attach(new ConcreteObserver(s,"Y"));
    s.Attach(new ConcreteObserver(s,"Z"));

    s.SubjectState = "ABC";
    s.Notify();

    Console.Read();
}
```

结果显示

观察者X的新状态是ABC
观察者Y的新状态是ABC
观察者Z的新状态是ABC

14.6　观察者模式特点

“用观察者模式的动机是什么呢？”小菜问道。

“问得好，将一个系统分割成一系列相互协作的类有一个很不好的副作用，那就是需要维护相关对象间的一致性。我们不希望为了维持一致性而使各类紧密耦合，这样会给维护、扩展和重用都带来不便**[DP]**。而观察者模式的关键对象是主题Subject和观察者Observer，一个Subject可以有任意数目的依赖它的Observer，一旦Subject的状态发生了改变，所有的Observer都可以得到通知。Subject发出通知时并不需要知道谁是它的观察者，也就是说，具体观察者是谁，它根本不需要知道。而任何一个具体观察者不知道也不需要知道其他观察者的存在。”

“什么时候考虑使用观察者模式呢？”

“你说什么时候应该使用？”大鸟反问道。

“当一个对象的改变需要同时改变其他对象的时候。”

“补充一下，而且它不知道具体有多少对象有待改变时，应该考虑使用观察者模式。还有吗？”

“我感觉当一个抽象模型有两个方面，其中一方面依赖于另一方面，这时用观察者模式可以将这两者封装在独立的对象中使它们各自独立地改变和复用。”

“非常好，总的来讲，观察者模式所做的工作其实就是在解除耦合。让耦合的双方都依赖于抽象，而不是依赖于具体。从而使得各自的变化都不会影响另一边的变化。”

“啊，这实在是依赖倒转原则的最佳体现呀。”小菜感慨道。

“我问你，在抽象观察者时，你的代码里用的是抽象类，为什么不用接口？”

“因为我觉得两个具体观察者，看股票观察者和看NBA观察者类是相似的，所以用了抽象类，这样可以共用一些代码，用接口只是方法上的实现，没什么太大意义。”

“那么抽象观察者可不可以用接口来定义？”

“用接口？我不知道，应该没必要吧。”

“哈，那是因为你不知道观察者模式的应用都是怎么样的。现实编程中，具体的观察者完全有可能是风马牛不相及的类，但它们都需要根据通知者的通知来做出Update（）的操作，所以让它们都实现下面这样的一个接口就可以实现这个想法了。”

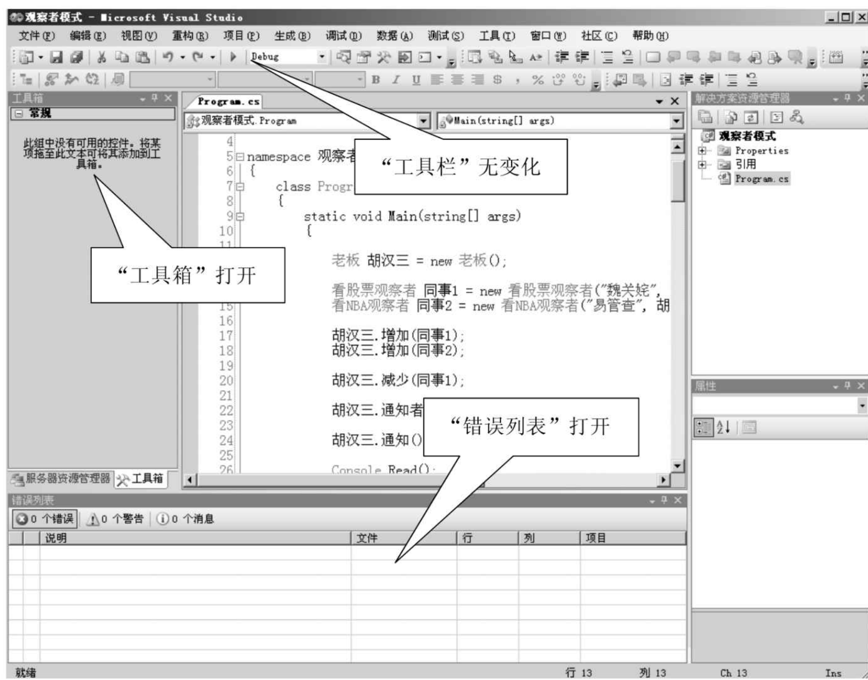
```
interface Observer
{
    void Update();
}
```

“嘿，大鸟说得好，这时用接口比较好。”小菜傻笑道。突然他表情一变，“等等，这里还是有问题，这些控件要么是.NET类库，要么是其他人事先写好的控件，它如何再能去实现拥有Update的Observer接口呢？”

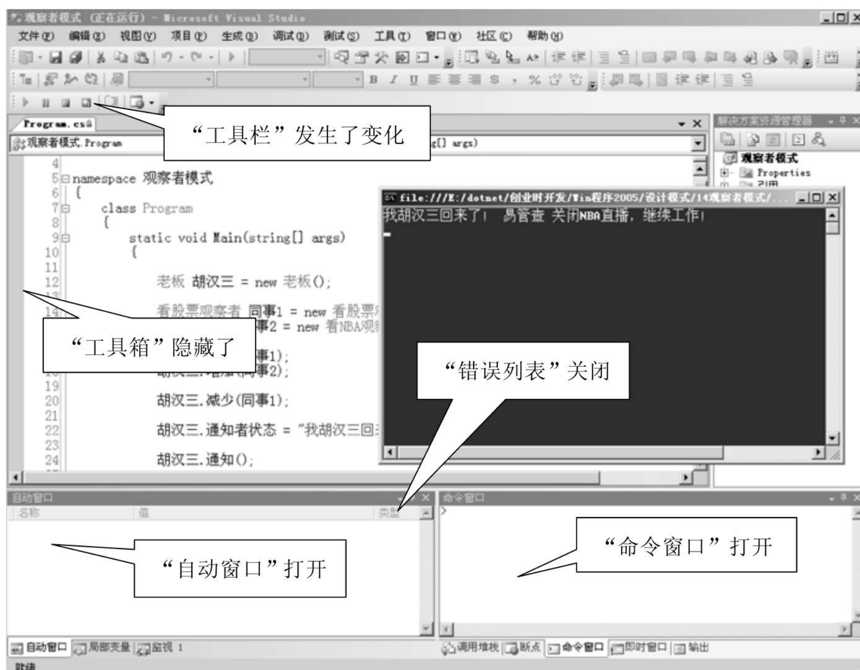
14.7 观察者模式的不足

大鸟微笑着说道："举个例子让我听听。"

"这样的例子很好举呀，"小菜说，"比如，VS 2005，当你点击运行程序之前的时候，整个界面是下面这样的。"



"当运行程序以后，除了弹出一个控制台的程序窗体以外，工具栏发生了变化，工具箱不见了，'错误列表'变成了'自动窗口'和'命令窗口'打开。仅仅是点击了一个'运行'按钮，就发生了这么多的变化，而各个变化都涉及到不同的控件。"



"我觉得没办法让每个控件都去实现一个'Observer'的接口，因为这些控件都早已被它们的制造商给封装了。"

"小菜聪明，你问到点子上了，尽管当点击'运行'按钮时，确实是在通知相关的控件产生变化，但它们是不可能用接口的方式来实现观察者模式的。"

"那怎么办呢？"

"是呀，那该怎么办呢？"

"凉拌呗。"大鸟得意之极。

"快说，摆什么架子！"

"还是回到刚才那个'老板、前台与同事的例子'，你看看它还有什么不足之处？"

"和刚才说的问题一样呀，尽管已经用了依赖倒转原则，但是'抽象通知者'还是依赖'抽象观察者'，也就是说，万一没有了抽象观察者这样的接口，我这通知的功能就完不成了。另外就是每个具体观察者，它不一定是'更新'的方法要调用呀，就像刚才说的，我希望的是'工具箱'是隐藏，'自动窗口'是打开，这根本就不是同名的方法。这应该就是不足的地方吧。"

“是呀，如果通知者和观察者之间根本就互相不知道，由客户端来决定通知谁，那就好了。来，我们先来把原来的代码改造一下。”

14.8 事件委托实现

“看股票观察者”类和“看NBA观察者”类，去掉了父类“抽象观察类”，所以补上一些代码，并将“更新”方法名改为各自适合的方法名。

```
//看股票的同事
class StockObserver
{
    private string name;
    private Subject sub;
    public StockObserver(string name, Subject sub)
    {
        this.name = name;
        this.sub = sub;
    }

    //关闭股票行情
    public void CloseStockMarket()
    {
        Console.WriteLine("{0} {1} 关闭股票行情，继续工作！", sub.SubjectState, name);
    }
}
```

方法“更新”名改为“关闭股票程序”

```
//看 NBA 的同事
class NBAObserver
{
    private string name;
    private Subject sub;
    public NBAObserver(string name, Subject sub)
    {
        this.name = name;
        this.sub = sub;
    }

    //关闭 NBA 直播
    public void CloseNBADirectSeeding()
    {
        Console.WriteLine("{0} {1} 关闭 NBA 直播，继续工作！", sub.SubjectState, name);
    }
}
```

方法“更新”名改为“关闭 NBA 直播”

“现实中就是这样的，方法名本就不一定相同。”小菜点头说。

“抽象通知者”由于不希望依赖“抽象观察者”，所以“增加”和“减少”的方法也就没有必要了（抽象观察者已经不存在了）。

```
//通知者接口
interface Subject
{
    void Notify();
    string SubjectState
```

```
        {
            get;
            set;
        }
    }
```

“下面就是如何处理‘老板’类和‘前台’类的问题，它们当中‘通知’方法有了对‘观察者’遍历，所以不可小视之。但如果在.NET中，我们可以用一个非常好的技术来处理这个问题，它叫……”

“快说。”小菜眼睛瞪大，盯着大鸟。

“它叫委托。”

“哦，就是委托呀，我都认真学过好几次了，但还是不太懂，同学中几个也都差不多，反正就不太明白，它到底是怎么回事，如何用。”

“先别管委托是怎么回事。我们看看如何做。”

声明一个委托，名称叫“EventHandler（事件处理程序）”，无参数，无返回值。

```
delegate void EventHandler();
```

“老板”类和“前台秘书”类

```
class Boss : Subject
{
    //声明一事件 Update，类型为委托 EventHandler
    public event EventHandler Update;

    private string action;

    public void Notify()
    {
        Update();
    }
    public string SubjectState
    {
        get { return action; }
        set { action = value; }
    }
}

class Secretary : Subject
{
    //与老板类类似，省略
}
```

声明一“EventHandler（事件处理程序）”的委托事件，名称叫“Update（更新）”

在访问“通知”方法时，调用“更新”

客户端代码

```
//老板胡汉三
Boss huhansan = new Boss();

//看股票的同事
StockObserver tongshi1 = new StockObserver("魏关姹", huhansan);
//看 NBA 的同事
NBAObserver tongshi2 = new NBAObserver("易管查", huhansan);

huhansan.Update += new EventHandler(tongshi1.CloseStockMarket);
huhansan.Update += new EventHandler(tongshi2.CloseNBADirectSeeding);

//老板回来
huhansan.SubjectState = "我胡汉三回来了！";
//发出通知
huhansan.Notify();
```

将“看股票者”的“关闭股票程序”方法和“看 NBA 者”的“关闭 NBA 直播”方法挂钩到“老板”的“更新”上，也就是将两不同类的不同方法委托给“老板”类的“更新”了

显示结果

我胡汉三回来了，魏关姹 关闭股票行情，继续工作！
我胡汉三回来了，易管查 关闭NBA直播，继续工作！

14.9 事件委托说明

“现在可以来解释一下，委托是什么了。委托就是一种引用方法的类型。一旦为委托分配了方法，委托将与该方法具有完全相同的行为。委托方法的使用可以像其他任何方法一样，具有参数和返回值。委托可以看作是对函数的抽象，是函数的‘类’，委托的实例将代表一个具体的函数。”

“哦，你的意思是说，‘delegate void EventHandler（）;’，可以理解为声明了一个特殊的‘类’。而‘public event EventHandler Update;’可以理解为声明了一个‘类’的变量。”

“哈哈，应该是声明了一个事件委托变量叫‘更新’。”

“你说的委托的实例将代表一个具体的函数，意思是说，‘new EventHandler（tongshi1.CloseStock Market）’其实就是一个委托的实例，而它就等于将‘tongshi1.CloseStockMarket’这个方法委托给‘huhansan.Update’这个方法了。”

“对了，就是这个意思，我刚才不是说过吗，一旦为委托分配了方法，委托将与该方法具有完全相同的行为。而且，一个委托可以搭载多个方法，所有方法被依次唤起。更重要的是，它可以使得委托对象所搭载的方法并不需要属于同一个类。你还不明白？”

“啊，我明白了，我明白了，”小菜脸笑开了花，“这样就使得，本来是在‘老板’类中的增加和减少的抽象观察者集合以及通知时遍历的抽象观察者都不必要了。转到客户端来让委托搭载多个方法，这就解决了本来与抽象观察者的耦合问题。”

“但委托也是有前提的，那就是委托对象所搭载的所有方法必须具有相同的原形和形式，也就是拥有相同的参数列表和返回值类型。”

“如果参数列表都不同那还瞎掺和啥。”小菜惊叹道，“太强了，当时那些牛人是怎么设计出这东西的，本来观察者模式已经把依赖倒转原则做到非常好了，现在看来，委托和事件，岂不是更加优秀，解决问题更加优雅？”

“注意，是先有观察者模式，再有委托事件技术的，再说，它们各有优缺点，你不妨去看看MSDN，讲得已经很详细了。”

“我现在对委托和事件有些了解了，相信再去研究MSDN也就不难了。这时再看看刚才举的那个例子，当点‘运行’时，所谓的‘工具箱隐藏’、‘错误列表隐藏’、‘自动窗口打开’、‘命令窗口打开’不过就是‘运行’时

注册的四个事件的触发而已。别说就四个，就是四十个完全不同的控件，也都能通知到位了。”

“夸张是有点夸张，不过确实也是如此。”

（注：委托和事件在附录一中的1.13节中还有讲解，可参考）

14.10　石守吉失手机后的委托

突然小菜的手机响了。

“小菜，我是石守吉，昨天我手机丢了，没办法，只得重买一个。原来的那个号也没法办回来，还好我记得你的手机，所以用这个新号打给你了。你能不能把我们班级同学的号码抄一份发邮件给我？”

“哦，这个好办，不过班级这么多人，我要是抄起来，也容易错。而且，如果现在同学有急事要找你，不就找不到了吗？我们这样办吧……用观察者模式。”

“你说什么？我听不懂呀。什么观察者模式？”

“哈，其实就是我在这里给我们班级所有同学群发一条短消息，通知他们，你石守吉已换新号，请大家更新号码，有事可及时与石守吉联系。”

“好办法，你可记得一定要给李MM、张MM、王MM发哦。”

“你小子，首先想着的就是MM。放心吧，我才不管谁呢，凡是在我手机里存的班级同学，我都会循环遍历一遍，群发给他们的。”

“小菜怎么张口闭口都是术语呀，好的，你就循环遍历一下吧。这事就委托给你了，谢谢哦！”

第15章　就不能不换DB吗？——抽象工厂模式

15.1　就不能不换DB吗？

时间：4月17日　23点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“这么晚才回来，都11点了。”大鸟看着刚推门而入的小菜问道。

“嗨，没办法呀，工作忙。”小菜叹气说道。

“怎么会这么忙，加班有点过头了呀。”

“都是换数据库惹的祸呗。”

“怎么了？”

“我本来写好了一个项目，是给一家企业做的电子商务网站，是用SQL　Server作为数据库的，应该说上线后除了开始有些小问题，基本都还可以。而后，公司接到另外一家公司类似需求的项目，但这家公司想省钱，租用了一个空间，只能用Access，不能用SQL　Server，于是就要求我今天改造原来那个项目的代码。”

“哈哈，你的麻烦来了。”

“是呀，那是相当的麻烦。但开始我觉得很简单呀，因为地球人都知道，SQL　Server和Access在ADO.NET上的使用是不同的，在SQL　Server上用的是System.Data.SqlClient命名空间下的SqlConnection、SqlCommand、SqlParameter、SqlDataReader、SqlDataAdapter，而Access则要用System.Data.OleDb命名空间下的相应对象，我以为只要做一个全体替换就可以了，哪知道，替换后，错误百出。”

“那是一定的，两者有不少不同的地方。你都找到了些什么问题？”

“实在是多呀。在插入数据时Access必须要insert　into而SQL　Server可以不用into的；SQL　Server中的GetDate（）在Access中没有，需要改成Now（）；SQL　Server中有字符串函数Substring，而Access中根本不能用，我找了很久才知道，可以用Mid，这好像是VB中的函数。”

“小菜还真犯了不少错呀，insert　into这是标准语法，你干吗不加into，这是自找的麻烦。”

“这些问题也就罢了，最气人的是程序的登录代码，老是报错，我怎么也找不到出了什么问题，搞了几个小时。最后才知道，原来Access对一些关键字，例如password是不能作为数据库的字段的，如果密码的字段名是password，SQL　Server中什么问题都没有，运行正常，在Access中就是报错，而且报得让人莫名其妙。”

“‘关键字’应该要用‘[’和‘]’包起来，不然当然是容易出错的。”

“就这样，今天加班到这时候才回来。”

“以后你还有的是班要加了。”

“为什么？”

“只要网站要维护，比如修改或增加一些功能，你就得改两个项目吧，至少在数据库中做改动，相应的程序代码都要改，甚至和数据库不相干的代码也要改，你既然有两个不同的版本，两倍的工作量也是必然的。”

“是呀，如果哪一天要用Oracle数据库，估计我要改动的地方更多了。”

“那是当然，Oracle的SQL语法与SQL　Server的差别更大。你的改动将是空前的。”

“大鸟只会夸张，哪有这么严重，大不了再加两天班就什么都搞定了。”

“哼”，大鸟笑着摇了摇头，很不屑一顾，“菜鸟程序员碰到问题，只会用时间来摆平，所以即使整天加班，老板也不想给菜鸟加工资，原因就在于此。”

“你什么意思嘛！”小菜气道，“我是菜鸟我怕谁。”接着又拉了拉大鸟，“那你说怎么搞定才是好呢？”

“知道求我啦，”大鸟端起架子，“教你可以，这一周的碗你洗。”

“行，”小菜很爽快地答应道，“在家洗碗也比加班熬夜强。”

15.2 最基本的数据访问程序

“你先写一段你原来的数据访问的做法给我看看。”

“那就用‘新增用户’和‘得到用户’为例吧。”

用户类，假设只有ID和Name两个字段，其余省略。

```
class User
{
    private int _id;
    public int ID
    {
        get {return _id;}
        set {_id=value;}
    }

    private string _name;
    public string Name
    {
        get { return _name; }
        set {_name=value;}
    }
}
```

SqlserverUser类——用于操作User表，假设只有“新增用户”和“得到用户”方法，其余方法以及具体的SQL语句省略。

```
class SqlserverUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在SQL Server中给User表
增加一条记录");
    }

    public User GetUser(int id)
    {
        Console.WriteLine("在SQL Server中根据ID得
```

到User表一条记录");
            return null;
        }
    }

客户端代码

```
static void Main(string[] args)
{
    User user = new User();

    SqlserverUser su = new SqlserverUser();

    su.Insert(user);

    su.GetUser(1);

    Console.Read();
}
```

与 SQL Server 耦合

插入用户

得到 ID 为 1 的用户

“我最开始就是这样写的，非常简单。”

“这里之所以不能换数据库，原因就在于SqlserverUser su = new SqlserverUser（）使得su这个对象被框死在SQL Server上了。如果这里是灵活的，专业点的说法，是多态的，那么在执行‘su.Insert（user）;’和‘su.GetUser（1）;’时就不用考虑是在用SQL Server还是在用Access。”

“我明白你意思了，你是希望我用‘工厂方法模式’来封装new SqlserverUser（）所造成的变化？”

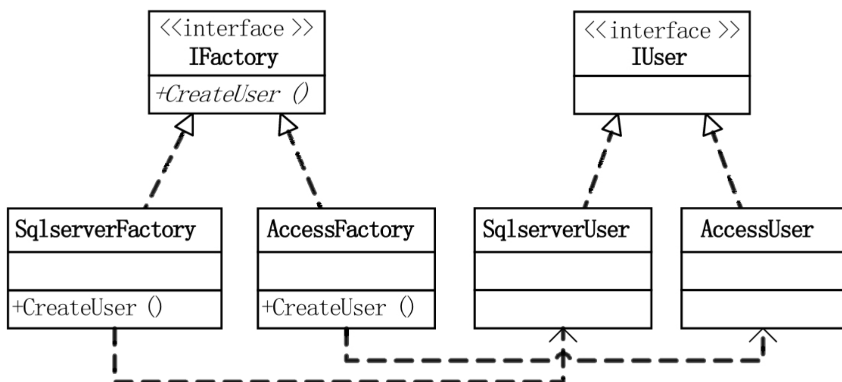
“小菜到了半夜，还是很清醒嘛，不错不错。”大鸟表扬道，“工厂方法模式是定义一个用于创建对象的接口，让子类决定实例化哪一个类。”接着说，“来试试看吧。”

“好。”

15.3 用了工厂方法模式的数据访问程序

小菜很快给出了工厂方法实现的代码。

代码结构图



IUser接口，用于客户端访问，解除与具体数据库访问的耦合。

```
interface IUser
{
    void Insert(User user);

    User GetUser(int id);
}
```

SqlserverUser类，用于访问SQL Server的User。

```
class SqlserverUser : IUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在SQL Server中给User表增加一条记录");
    }
```

```
    public User GetUser(int id)
    {
        Console.WriteLine("在SQL Server中根据ID得到User表一条记录");
        return null;
    }
}
```

AccessUser类，用于访问Access的User。

```
class AccessUser : IUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在Access中给User表增加一条记录");
    }

    public User GetUser(int id)
    {
        Console.WriteLine("在Access中根据ID得到User表一条记录");
        return null;
    }
}
```

IFactory接口，定义一个创建访问User表对象的抽象的工厂接口。

```
interface IFactory
{
    IUser CreateUser();
}
```

SqlServerFactory类，实现IFactory接口，实例化SqlserverUser。

```
class SqlServerFactory : IFactory
{
    public IUser CreateUser()
    {
        return new SqlserverUser();
    }
}
```

AccessFactory类，实现IFactory接口，实例化AccessUser。

```
class AccessFactory : IFactory
{
    public IUser CreateUser()
    {
        return new AccessUser();
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    User user = new User();

    IFactory factory = new SqlServerFactory();

    IUser iu = factory.CreateUser();

    iu.Insert(user);
    iu.GetUser(1);

    Console.Read();
}
```

若要更改成 Access 数据库，只需要将本句改成 IFactory factory = new AccessFactory();

“大鸟，来看看这样写对不对？”

“非常好。现在如果要换数据库，只需要把new SqlServerFactory（）改成new　AccessFactory（），此时由于多态的关系，使得声明IUser接口的对象iu事先根本不知道是在访问哪个数据库，却可以在运行时很好地完成工作，这就是所谓的业务逻辑与数据访问的解耦。”

“但是，大鸟，这样写，代码里还是有指明‘new SqlServerFactory（）’呀，我要改的地方，依然很多。”

“这个先不急，待会再说，问题没有完全解决，你的数据库里不可能

只有一个User表吧，很可能有其他表，比如增加部门表（Department表），此时如何办呢？”

```
class Department
{
    private int_id;
    public int ID
    {
        get{teturn_id;}
        set{_id=value;}
    }
    private string_deptName;
    public string DeptName
    {
        get{return_deptName;}
        set{_deptName=value;}
    }
}
```

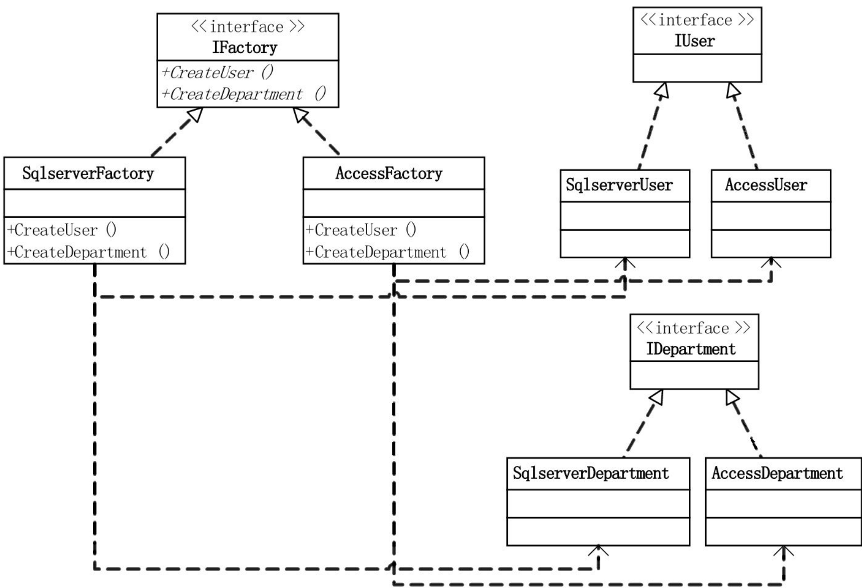
“啊，我觉得那要增加好多类了，我来试试看。”

“多写些类有什么关系，只要能增加灵活性，以后就不用加班了。小菜好好加油。”

15.4　用了抽象工厂模式的数据访问程序

小菜再次修改代码，增加了关于部门表的处理。

代码结构图



IDepartment接口，用于客户端访问，解除与具体数据库访问的耦合。

```
interface IDepartment
{
    void Insert(Department department);

    Department GetDepartment(int id);
}
```

SqlserverDepartment类，用于访问SQL Server的Department。

```
class SqlserverDepartment : IDepartment
```

```
{
    public void Insert(Department department)
    {
        Console.WriteLine("在SQL Server中给
Department表增加一条记录");
    }

    public Department GetDepartment(int id)
    {
        Console.WriteLine("在SQL Server中根据ID得
到Department表一条记录");
        return null;
    }
}
```

AccessDepartment类，用于访问Access的Department。

```
class AccessDepartment : IDepartment
{
    public void Insert(Department department)
    {
        Console.WriteLine("在Access中给Department
表增加一条记录");
    }

    public Department GetDepartment(int id)
    {
        Console.WriteLine("在Access中根据ID得到
Department表一条记录");
        return null;
    }
}
```

IFactory接口，定义一个创建访问Department表对象的抽象的工厂接口。

```
interface IFactory
{
    IUser CreateUser();
    IDepartment CreateDepartment();
}
```

增加的接口方法

SqlServerFactory类，实现IFactory接口，实例化SqlserverUser和SqlserverDepartment。

```
class SqlServerFactory :IFactory
{
    public IUser CreateUser()
    {
        return new SqlserverUser();
    }
    public IDepartment CreateDepartment()
    {
        return new SqlserverDepartment();
    }
}
```

增加了 SqlserverDepartment 工厂

AccessFactory类，实现IFactory接口，实例化AccessUser和AccessDepartment。

```
class AccessFactory : IFactory
{
    public IUser CreateUser()
    {
        return new AccessUser();
    }
    public IDepartment CreateDepartment()
    {
        return new AccessDepartment();
    }
}
```

增加了 AccessDepartment 工厂

客户端代码

```
static void Main(string[] args)
{
    User user = new User();
    Department dept = new Department();

    //IFactory factory = new SqlServerFactory();
```

只需确定实例化哪一个数据库访问对象给 factory

```
        IFactory factory = new AccessFactory();

        IUser iu = factory.CreateUser();

        iu.Insert(user);
        iu.GetUser(1);

        IDepartment id = factory.CreateDepartment();
        id.Insert(dept);
        id.GetDepartment(1);

        Console.Read();
    }
```

则此时已与具体的数据库访问解除了依赖

则此时已与具体的数据库访问解除了依赖

结果显示如下：

```
在Access中给User表增加一条记录
在Access中根据ID得到User表一条记录
在Access中给Department表增加一条记录
在Access中根据ID得到Department表一条记录
```

“大鸟，这样就可以做到，只需更改IFactory factory = new AccessFactory（）为IFactory factory = new SqlServerFactory（），就实现了数据库访问的切换了。”

“很好，实际上，在不知不觉间，你已经通过需求的不断演化，重构出了一个非常重要的设计模式。”

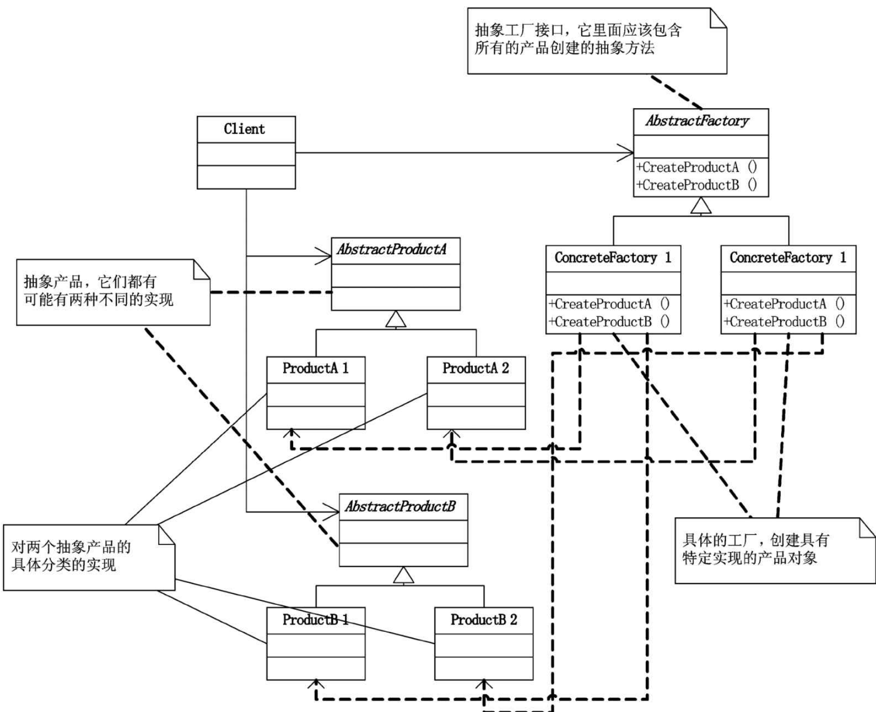
“刚才不就是工厂方法模式吗？”

“只有一个User类和User操作类的时候，是只需要工厂方法模式的，但现在显然你数据库中有很多的表，而SQL Server与Access又是两大不同的分类，所以解决这种涉及到多个产品系列的问题，有一个专门的工厂模式叫抽象工厂模式。”

15.5 抽象工厂模式

> 抽象工厂模式（**Abstract Factory**），提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。[DP]

抽象工厂模式（**Abstract Factory**）结构图



“**AbstractProductA**和**AbstractProductB**是两个抽象产品，之所以为抽象，是因为它们都有可能有两种不同的实现，就刚才的例子来说就是User和Department，而**ProductA1**、**ProductA2**和**ProductB1**、**ProductB2**就是对两个抽象产品的具体分类的实现，比如ProductA1可以理解为是SqlserverUser，而ProductB1是SqlserverDepartment。”

“这么说，**IFactory**是一个抽象工厂接口，它里面应该包含所有的产品创建的抽象方法。而**ConcreteFactory1**和**ConcreteFactory2**就是具体的工厂了。就像SqlserverFactory和AccessFactory一样。”

“理解得非常正确。通常是在运行时刻再创建一个**ConcreteFactory**类的实例，这个具体的工厂再创建具有特定实现的产品对象，也就是

说，为创建不同的产品对象，客户端应使用不同的具体工厂。”

15.6 抽象工厂模式的优点与缺点

“这样做的好处是什么呢？”

“最大的好处便是易于交换产品系列，由于具体工厂类，例如IFactory factory = new AccessFactory（），在一个应用中只需要在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得非常容易，它只需要改变具体工厂即可使用不同的产品配置。我们的设计不能去防止需求的更改，那么我们的理想便是让改动变得最小，现在如果你要更改数据库访问，我们只需要更改具体工厂就可以做到。第二大好处是，它让具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操纵实例，产品的具体类名也被具体工厂的实现分离，不会出现在客户代码中。事实上，你刚才写的例子，客户端所认识的只有IUser和IDepartment，至于它是用SQL Server来实现还是Access来实现就不知道了。”

“啊，我感觉这个模式把开放-封闭原则，依赖倒转原则发挥到极致了。”小菜说。

“没这么夸张，应该说就是这些设计原则的良好运用。抽象工厂模式也有缺点。你想得出来吗？”

“想不出来，我觉得它已经很好用了，哪有什么缺点。”

“是个模式都是会有缺点的，都有不适用的时候，要辨证地看待问题哦。抽象工厂模式可以很方便地切换两个数据库访问的代码，但是如果你的需求来自增加功能，比如我们现在要增加项目表Project，你需要改动哪些地方？”

“啊，那就至少要增加三个类，**IProject**、**SqlserverProject**、**AccessProject**，还需要更改**IFactory**、**SqlserverFactory**和**AccessFactory**才可以完全实现。啊，要改三个类，这太糟糕了。”

“是的，这非常糟糕。”

“还有就是刚才问你的，我的客户端程序类显然不会是只有一个，有很多地方都在使用**IUser**或**IDepartment**，而这样的设计，其实在每一个类的开始都需要声明IFactory factory = new SqlserverFactory（），如果我有100个调用数据库访问的类，是不是就要更改100次IFactory factory = new AccessFactory（）这样的代码才行？这不能解决我要更改数据库访问时，改动一处就完全更改的要求呀！”

“改就改啰，公司花这么多钱养你干吗？不就是要你努力工作吗。100个改动，不算难的，加个班，什么都搞定了。”大鸟一脸坏笑地说

道。

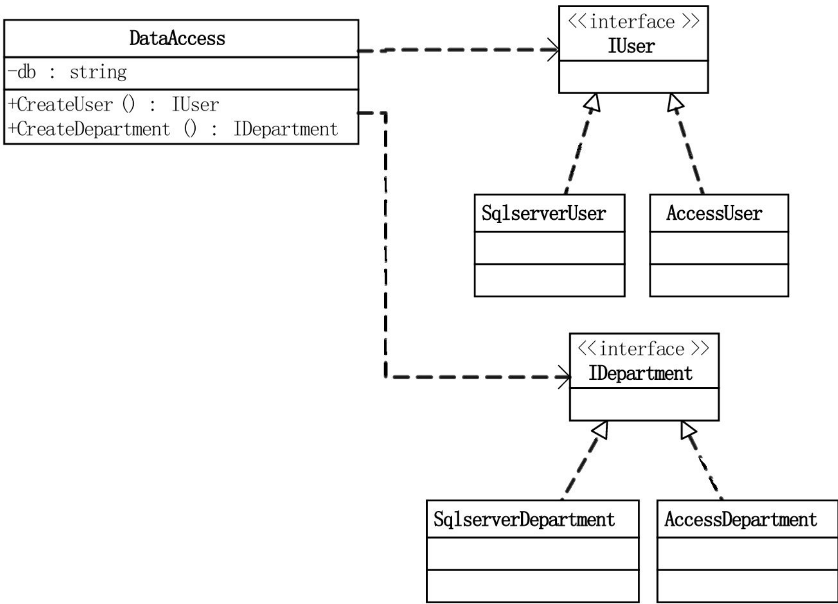
“不可能，你讲过，编程是门艺术，这样大批量的改动，显然是非常丑陋的做法。一定有更好的办法。”小菜非常肯定地答道，“我来想想办法改进一下这个抽象工厂。”

“好，小伙子，有立场，有想法，不向丑陋代码低头，那就等你的好消息。”大鸟点头肯定。

15.7　用简单工厂来改进抽象工厂

十分钟后，小菜给出了一个改进方案。去除 **IFactory**、**SqlserverFactory**和**AccessFactory**三个工厂类，取而代之的是**DataAccess**类，用一个简单工厂模式来实现。

代码结构图




```
class DataAccess
{
    private static readonly string db = "Sqlserver";
    //private static readonly string db = "Access";

    public static IUser CreateUser()
    {
        IUser result=null;
        switch (db)
        {
            case "Sqlserver":
                result = new SqlserverUser();
                break;
            case "Access":
                result = new AccessUser();
                break;
        }
        return result;
    }

    public static IDepartment CreateDepartment()
    {
        IDepartment result = null;
        switch (db)
```

数据库名称，可替换成 Access

由于 db 的事先设置，所以此处可以根据选择实例化出相应的对象

```
            {
                case "Sqlserver":
                    result = new SqlserverDepartment();
                    break;
                case "Access":
                    result = new AccessDepartment();
                    break;
            }
            return result;
        }
    }
```

客户端代码

```
static void Main(string[] args)
{
    User user = new User();
    Department dept = new Department();

    IUser iu = DataAccess.CreateUser();

    iu.Insert(user);
    iu.GetUser(1);

    IDepartment id = DataAccess.CreateDepartment();
    id.Insert(dept);
    id.GetDepartment(1);

    Console.Read();
}
```

直接得到实际的数据库访问实例，而不存在任何依赖

直接得到实际的数据库访问实例，而不存在任何依赖

“大鸟，来看看我的设计，我觉得这里与其用那么多工厂类，不如直接用一个简单工厂来实现，我抛弃了**IFactory**、**SqlserverFactory**和**AccessFactory**三个工厂类，取而代之的是**DataAccess**类，由于事先设置了db的值（**Sqlserver**或**Access**），所以简单工厂的方法都不需要输入参数，这样在客户端就只需要DataAccess.CreateUser（）和DataAccess.CreateDepartment（）来生成具体的数据库访问类实例，客户端没有出现任何一个**SQL Server**或**Access**的字样，达到了解耦的目的。”

“哈，小菜，厉害厉害，你的改进确实是比之前的代码要更进一步了，客户端已经不再受改动数据库访问的影响了。可以打95分。”大鸟拍了拍小菜，鼓励地说，“为什么不能得满分，原因是如果我需要增加**Oracle**数据库访问，本来抽象工厂只增加一个OracleFactory工厂类就可

以了，现在就比较麻烦了。”

“是的，没办法，这样就需要在**DataAccess**类中每个方法的**swicth**中加**case**了。”

15.8　用反射+抽象工厂的数据访问程序

“我们要考虑的就是可不可以不在程序里写明‘如果是Sqlserver就去实例化SQL　Server数据库相关类，如果是Access就去实例化Access相关类’这样的语句，而是根据字符串db的值去某个地方找应该要实例化的类是哪一个。这样，我们的switch就可以对它说再见了。”

“听不太懂哦，什么叫‘去某个地方找应该要实例化的类是哪一个’？”小菜糊涂地问。

“我要说的就是一种编程方式：依赖注入（Dependency Injection），从字面上不太好理解，我们也不去管它。关键在于如何去用这种方法来解决我们的switch问题。本来依赖注入是需要专门的IoC容器提供，比如Spring.NET，显然当前这个程序不需要这么麻烦，你只需要再了解一个简单的.NET技术‘反射’就可以了。”

“大鸟，你一下子说出又是‘依赖注入’又是‘反射’这些莫名其妙的名词，很晕。”小菜有些犯困，“我就想知道，如何向switch说bye-bye！至于那些什么概念我不想了解。”

“心急讨不了好媳妇！你急什么？”大鸟嘲笑道，“反射技术看起来很玄乎，其实实际用起来不算难。它的格式是

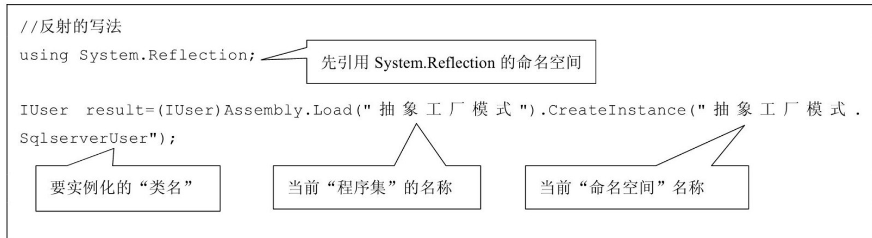
Assembly.Load("程序集名称").CreateInstance("命名空间.类名称")

只要在程序顶端写上using System.Reflection;来引用Reflection，就可以使用反射来帮我们克服抽象工厂模式的先天不足了。”

“具体怎么做呢？快说快说。”小菜有些着急。

“有了反射，我们获得实例可以用下面两种写法。”

```
//常规的写法
IUser result = new SqlserverUser();
```



“实例化的效果是一样的，但这两种方法的区别在哪里？”大鸟问道。

“常规方法是写明了要实例化SqlserverUser对象。反射的写法，其实也是指明了要实例化SqlserverUser对象呀。”

“常规方法你可以灵活更换为AccessUser吗？”

“不可以，这都是事先编译好的代码。”

“那你看看，在反射中‘CreateInstance（"抽象工厂模式. SqlserverUser"）’，可以灵活更换‘SqlserverUser’为‘AccessUser’吗？”

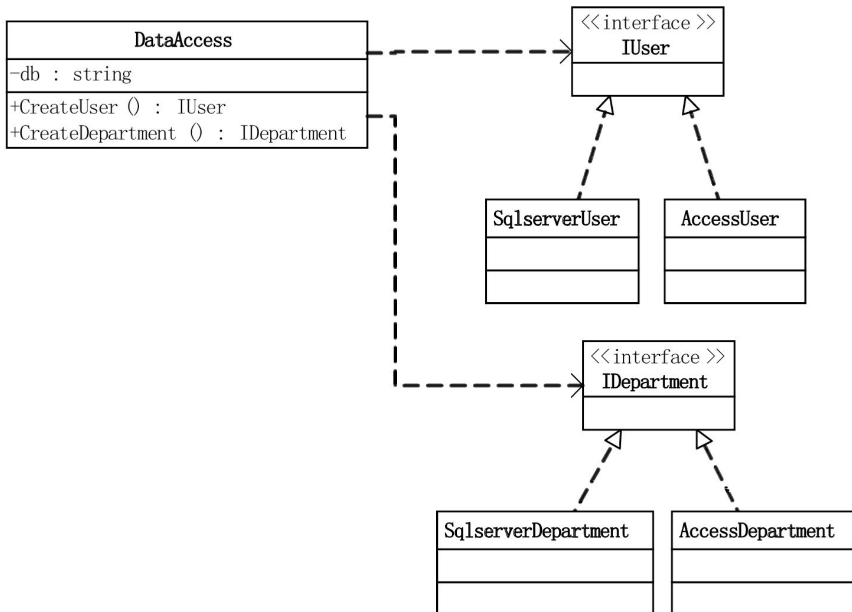
“还不是一样，写死在代码…………等等，哦！！！我明白了。”小菜一下子顿悟过来，兴奋起来。“因为这里是字符串，可以用变量来处理，也就可以根据需要更换。哦，My God！太妙了！”

“哈哈，我以前对你讲四大发明之活字印刷时，曾说过‘体会到面向对象带来的好处，那种感觉应该就如同是一中国酒鬼第一次喝到了茅台，西洋酒鬼第一次喝到了XO一样，怎个爽字可形容呀’，你有没有这种感觉了？”

“嗯，我一下子知道这里的差别主要在原来的实例化是写死在程序里的，而现在用了反射就可以利用字符串来实例化对象，而变量是可以更换的。”小菜说道。

“写死在程序里，太难听了。准确地说，是将程序由编译时转为运行时。由于‘CreateInstance（"抽象工厂模式. SqlserverUser"）’中的字符串是可以写成变量的，而变量的值到底是SQL Server，还是Access，完全可以由事先的那个db变量来决定。所以就去除了switch判断的麻烦。”

代码结构图



DataAccess类，用反射技术，取代**IFactory**、**SqlserverFactory**和**AccessFactory**。

```
using System.Reflection;

class DataAccess
{
    private static readonly string AssemblyName = "抽象工厂模式";
    private static readonly string db = "Sqlserver";

    public static IUser CreateUser()
    {
        string className = AssemblyName + "." + db + "User";
        return (IUser)Assembly.Load(AssemblyName).CreateInstance(className);
    }
    public static IDepartment CreateDepartment()
    {
        string className = AssemblyName + "." + db + "Department";
        return (IDepartment)Assembly.Load(AssemblyName).CreateInstance(className);
    }
}
```

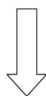
引入反射，必须要写

程序集名称

数据库名称，可替换成 Access

“现在如果我们增加了Oracle数据访问，相关的类的增加是不可避免的，这点无论我们用任何办法都解决不了，不过这叫扩展，开放-封闭原则性告诉我们，对于扩展，我们开放。但对于修改，我们应该要尽量关闭，就目前而言，我们只需要更改private static readonly string db = "Sqlserver";为private static readonly string db = "Oracle";也就意味着（IUser）Assembly.Load（AssemblyName）.CreateInstance（className）;这一句发生了变化。”

```
return (IUser)Assembly.Load("抽象工厂模式").CreateInstance("抽象工厂模式.SqlserverUser");
```



```
return (IUser)Assembly.Load("抽象工厂模式").CreateInstance("抽象工厂模式.OracleUser");
```

“这样的结果就是DataAccess.CreateUser（）本来得到的是SqlserverUser的实例，而现在变成了OracleUser的实例了。”

“那么如果我们需要增加Project产品时，如何做呢？”

“只需要增加三个与Project相关的类，再修改DataAccesss，在其中增加一个public static IProject CreateProject（）方法就可以了。”

“怎么样，编程的艺术感是不是出来了？”

“哈，比以前，这代码是漂亮多了。但是，总感觉还是有点缺憾，因为在更换数据库访问时，我还是需要去改程序（改db这个字符串的值）重编译，如果可以不改程序，那才是真正地符合开放-封闭原则。”

15.9 用反射+配置文件实现数据访问程序

“小菜很追求完美嘛！我们还可以利用配置文件来解决更改DataAccess的问题的。”

“哦，对的，对的，我可以读文件来给DB字符串赋值，在配置文件中写明是Sqlserver还是Access，这样就连DataAccess类也不用更改了。”

添加一个App.config文件。内容如下。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
      <add key="DB" value="Sqlserver"/>
  </appSettings>
</configuration>
```

配置文件，可以换成Access 或 Oracle

再添加引用System.configuration，并在程序开头增加using System.Configuration;，然后更改DataAccess类的字段DB的赋值代码。

```
private static readonly string db = ConfigurationManager.AppSettings["DB"];
```

表示读配置文件

“哈，这下基本可以算是满分了，现在我们应用了反射+抽象工厂模式解决了数据库访问时的可维护、可扩展的问题。”

“从这个角度上说，所有在用简单工厂的地方，都可以考虑用反射技术来去除**switch**或**if**，解除分支判断带来的耦合。”

“说得没错，switch或者if是程序里的好东西，但在应对变化上，却显得老态龙钟。反射技术的确可以很好地解决它们难以应对变化，难以维护和扩展的诟病。”

15.10 无痴迷，不成功

“设计模式真的很神奇哦，如果早先是这样设计的话，我今天就用不着加班加点了。”

“好了，都快1点了，你还要不要睡觉呢？”

“啊，今天都加了一晚上的班，但学起设计模式来，我把时间都给忘记了，什么劳累都没了。”

“这就说明你是做程序员的料，一个程序员如果从来没有熬夜写程序的经历，不能算是一个好程序员，因为他没有痴迷过，所以他不会有大成就。”

“是的，无痴迷，不成功。我一定会成为优秀的程序员。我坚信。”小菜非常自信地说道。

第16章　无尽加班何时休——状态模式

16.1　加班，又是加班！

时间：4月19日　23点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“小菜，你们的加班没完没了了？”大鸟为晚上十点才到家的小菜打开了房门。

“嗨，没办法，公司的项目很急，所以要求要加班。”

“有这么急吗？这星期四天来你都在加班，有加班费吗？难道周末也要继续？”

“哪来什么加班费，周末估计是逃不了了。”小菜显然很疲惫，“经理把每个人每天的工作都排得满满的，说做完就可以回家，但是没有任何一个人可以在下班前完成的，基本都得加班，这就等于是自愿加班。我走时还有哥们在加班呢。”

“再急也不能这样呀，长时间加班，又没有加班费，士气更加低落，效率大打折扣。”

“可不是咋地！上午刚上班的时候，效率很高，可以写不少代码，到了中午，午饭一吃完，就犯困，可能是最近太累了，但还不敢休息，因为没有人趴着睡觉的，都说项目急，要抓紧。所以我就这么迷迷糊糊的，到了下午三点多才略微精神点，本想着今天任务还算可以，希望能早点完成，争取不要再加班了。哪知快下班时才发现有一个功能是我理解有误，其实比想象的要复杂得多。嗨！苦呀，又多花了三个多钟头，九点多才从公司出来。”

“哈，那你自己也有问题，对工作量的判断有偏差。在公司还可以通过加班来补偿，要是在高考考场上，哪可能加时间，做不完直接就是玩完。”

“你说这老板对加班是如何想的呢？难道真的认为加班可以解决问题？我感觉这样赶进度，对代码质量没任何好处。”

“老板的想法当然是和员工不一样了。员工加班，实际上分为几种，第一，极有可能是员工为了下班能多上会网，聊聊天，打打游戏，或者是为了学习点新东西，所以这其实根本就不能算是加班，只能算下班时坐在办公座位上，第二种，可能这个员工能力相对差，技术或业务能力不过关，或者动作慢，效率低，那当然应该要加班，而且老板也不会打算给这种菜鸟补偿。”

“大鸟，讽刺我呀。”小菜有些不满。

“我又没说是指你，除非你真的觉得自己能力差、效率低，是菜鸟。”

“不过也不得不承认，我现在经验不足确实在效率上是会受些影响的，公司里的一些骨灰级程序员，也不觉得水平特别厉害，但是总是能在下班前后就完成当天任务，而且错误很少。”

“慢慢来吧，编程水平也不是几天就可以升上去的。虽然今天你很累了，但是通过加班这件事，你也可以学到设计模式。”

“哦，听到设计模式，我就不感觉累了。来，说说看。”

“你刚才曾讲到，上午状态好，中午想睡觉，下午渐恢复，加班苦煎熬。其实是一种状态的变化，不同的时间，会有不同的状态。你现在用代码来实现一下。”

“其实就是根据时间的不同，做出判断来实现，是吧？这不是大问题。”

16.2 工作状态-函数版

半小时后，小菜的第一版程序。

```
static int Hour = 0; //钟点
static bool WorkFinished = false; //任务完成标记
//写程序方法
public static void WriteProgram()
{
    if (Hour < 12)
    {
        Console.WriteLine("当前时间：{0}点 上午工作，精神百倍", Hour);
    }
    else if (Hour < 13)
    {
        Console.WriteLine("当前时间：{0}点 饿了，午饭；犯困，午休。", Hour);
    }
    else if (Hour < 17)
    {
        Console.WriteLine("当前时间：{0}点 下午状态还不错，继续努力", Hour);
    }
    else
    {
        if (WorkFinished)
        {
            Console.WriteLine("当前时间：{0}点 下班回家了", Hour);
        }
        else
        {
            if (Hour < 21)
            {
                Console.WriteLine("当前时间：{0}点 加班哦，疲累之极", Hour);
            }
            else
            {
                Console.WriteLine("当前时间：{0}点 不行了，睡着了。", Hour);
            }
        }
    }
}
```

定义一个“写程序”的函数，用来根据时间的不同体现不同的工作状态

主程序如下：

```
static void Main(string[] args)
{
    Hour = 9;
    WriteProgram();
    Hour= 10;
    WriteProgram();
    Hour= 12;
    WriteProgram();
    Hour= 13;
    WriteProgram();
    Hour= 14;
    WriteProgram();
    Hour= 17;

    WorkFinished = true;
    //WorkFinished = false;

    WriteProgram();
    Hour= 19;
    WriteProgram();
    Hour= 22;
    WriteProgram();

    Console.Read();
}
```

任务完成，则可以下班，否则就得加班了

“小菜，都学了这么长时间的面向对象开发，你怎么还在写面向过程的代码呀？”

“啊，我习惯性思维了，你意思是说要分一个类出来。”

“这是起码的面向对象思维呀，至少应该有个“工作”类，你的‘写程序’方法是类方法，而‘钟点’、‘任务完成’其实就是类的什么？”

“应该是对外属性，是吧？”

“问什么问，还不快去重写。”大鸟不答反而催促道。

16.3 工作状态-分类版

十分钟后小菜写出了第二版程序。

工作类

```
public class Work
{
    private int hour;
    public int Hour
    {
        get { return hour; }
        set { hour = value; }
    }
    private bool finish = false;
    public bool TaskFinished
    {
        get { return finish; }
        set { finish = value; }
    }
    public void WriteProgram()
    {
        if(hour < 12)
        {
            Console.WriteLine("当前时间：{0}点 上午
工作，精神百倍",hour);
        }
        else if(hour < 13)
        {
           Console.WriteLine("当前时间：{0}点 饿
了，午饭；犯困，午休。",hour);
        }
        else if(hour < 17)
        {
           Console.WriteLine("当前时间：{0}点 下午
状态还不错，继续努力",hour);    }
        else
        {
            if(finish) {
                                        Console.WriteLine("当
前时间：{0}点 下班回家了",hour);      }
```

```
            else
            {
                if(hour < 21)
                {
                    Console.WriteLine("当前时间：
{0}点 加班哦，疲累之极",hour);
                }
                else
                {
                    Console.WriteLine("当前时间：
{0}点 不行了，睡着了。",hour);
                }
            }
        }
    }
}
```

客户端程序如下：

```
static void Main(string[] args)
{
    //紧急项目
    Work emergencyProjects = new Work();
    emergencyProjects.Hour = 9;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 10;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 12;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 13;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 14;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 17;

    //emergencyProjects.WorkFinished = true;
    emergencyProjects.TaskFinished = false;

    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 19;
    emergencyProjects.WriteProgram();
```

```
    emergencyProjects.Hour = 22;
    emergencyProjects.WriteProgram();

    Console.Read();
}
```

结果表现如下

当前时间：9点 上午工作，精神百倍
当前时间：10点 上午工作，精神百倍
当前时间：12点 饿了，午饭；犯困，午休
当前时间：13点 下午状态还不错，继续努力
当前时间：14点 下午状态还不错，继续努力
当前时间：17点 加班哦，疲累之极
当前时间：19点 加班哦，疲累之极
当前时间：22点 不行了，睡着了。

16.4　方法过长是坏味道

“若是‘任务完成’，则17点、19点、22点都是‘下班回家了’的状态了。”

“好，现在我来问你，这样的代码有什么问题？”大鸟问道。

“我觉得没什么问题呀，不然我早改了。”

“仔细看看，MartinFowler曾在《重构》中写过一个很重要的代码坏味道，叫做‘Long　Method’，方法如果过长其实极有可能是有坏味道了。”

“你的意思是‘Work（工作）’类的‘WriteProgram（写程序）’方法过长了？不过这里面太多的判断，好像是不太好。但我也想不出来有什么办法解决它。”

“你要知道，你这个方法很长，而且有很多的判断分支，这也就意味着它的责任过大了。无论是任何状态，都需要通过它来改变，这实际上是很糟糕的。”

“哦，对的，面向对象设计其实就是希望做到代码的责任分解。这个类违背了‘单一职责原则’。但如何做呢？”

“说得不错，由于‘WriteProgram（写程序）’的方法里有这么多判断，使得任何需求的改动或增加，都需要去更改这个方法了，比如，你们老板也感觉加班有些过分，对于公司的办公室管理以及员工的安全都不利，于是发了一通知，不管任务再多，员工必须在20点之前离开公司。这样的需求很合常理，所以要满足需求你就得更改这个方法，但真正要更改的地方只涉及到17点到22点之间的状态，但目前的代码却是对整个方法做改动的，维护出错的风险很大。”

“你解释了这么多，我的理解其实就是这样写方法违背了‘开放-封闭原则’。”

“哈，小菜总结得好，对这几个原则理解得很透嘛。那么我们应该如何做？”

“把这些分支想办法变成一个又一个的类，增加时不会影响其他类。然后状态的变化在各自的类中完成。”小菜说道，“理论讲讲很容易，但实际如何做，我想不出来。”

“当然，这需要丰富的经验积累，但实际上你是用不着再去重复发

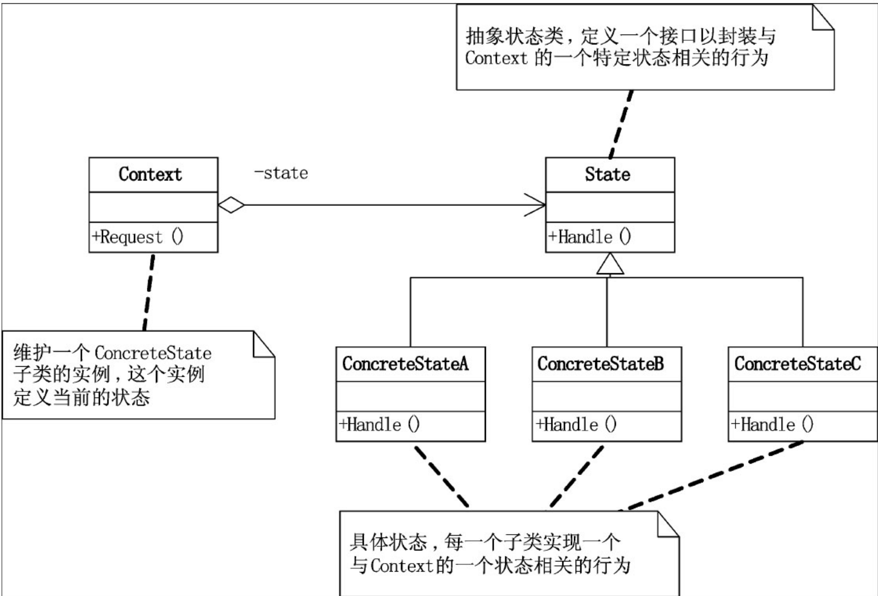
明‘轮子’了，因为GoF已经为我们针对这类问题提供了解决方案，那就是‘状态模式’。”

16.5 状态模式

> 状态模式（State），当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。[DP]

“状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类当中，可以把复杂的判断逻辑简化。当然，如果这个状态判断很简单，那就没必要用‘状态模式’了。”

状态模式（**State**）结构图



State类，抽象状态类，定义一个接口以封装与Context的一个特定状态相关的行为。

```
abstract class State
{
    public abstract void Handle(Context context);
}
```

ConcreteState类，具体状态，每一个子类实现一个与Context的一个状态相关的行为。

```
class ConcreteStateA : State
{
    public override void Handle(Context context)
    {
        context.State = new ConcreteStateB();
    }
}

class ConcreteStateB : State
{
    public override void Handle(Context context)
    {
        context.State = new ConcreteStateA();
    }
}
```

设置 ConcreteStateA 的下一状态是 ConcreteStateB

设置 ConcreteStateB 的下一状态是 ConcreteStateA

Context类，维护一个ConcreteState子类的实例，这个实例定义当前的状态。

```
class Context
{
    private State state;
    public Context(State state)
    {
        this.state = state;
    }

    public State State
    {
        get {  return state; }
        set
        {
            state = value;
            Console.WriteLine("当前状态:" + state.GetType().Name);
        }
    }

    public void Request()
    {
        state.Handle(this);
    }
}
```

定义 Context 的初始状态

可读写的状态属性，用于读取当前状态和设置新状态

对请求做处理，并设置下一状态

客户端代码

```
static void Main(string[] args)
{
    Context c = new Context(new ConcreteStateA());

    c.Request();
    c.Request();
    c.Request();
    c.Request();

    Console.Read();
}
```

16.6 状态模式好处与用处

“状态模式的好处是将与特定状态相关的行为局部化，并且将不同状态的行为分割开来[DP]。”

“是不是就是将特定的状态相关的行为都放入一个对象中，由于所有与状态相关的代码都存在于某个**ConcreteState**中，所以通过定义新的子类可以很容易地增加新的状态和转换 **[DP]** 。”

“说白了，这样做的目的就是为了消除庞大的条件分支语句，大的分支判断会使得它们难以修改和扩展，就像我们最早说的刻版印刷一样，任何改动和变化都是致命的。状态模式通过把各种状态转移逻辑分布到**State**的子类之间，来减少相互间的依赖，好比把整个版面改成了一个又一个的活字，此时就容易维护和扩展了。”

“什么时候应该考虑使用状态模式呢？”

“当一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑使用状态模式了。另外如果业务需求某项业务有多个状态，通常都是一些枚举常量，状态的变化都是依靠大量的多分支判断语句来实现，此时应该考虑将每一种业务状态定义为一个State的子类。这样这些对象就可以不依赖于其他对象而独立变化了，某一天客户需要更改需求，增加或减少业务状态或改变状态流程，对你来说都是不困难的事。”

“哦，明白了，这种需求还是非常常见的。”

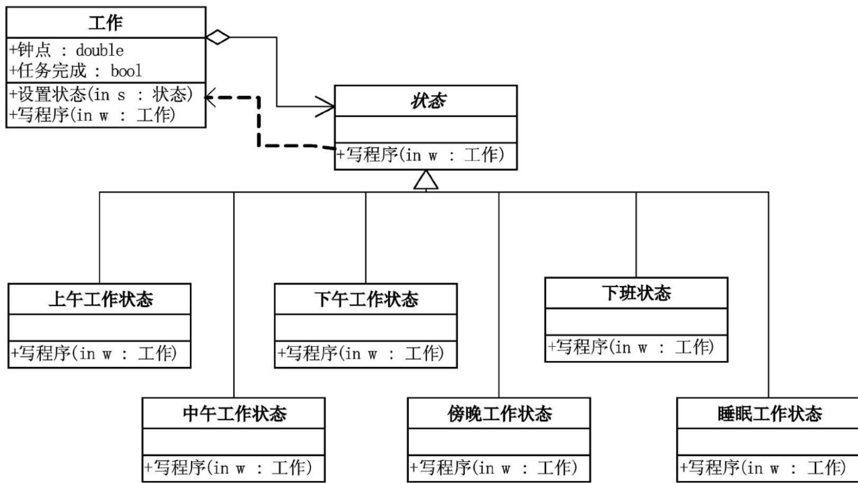
“现在再回过头来看你的代码，那个‘Long Method’你现在会改了吗？”

“哦，学了状态模式，有点感觉了，我试试看。”

16.7 工作状态-状态模式版

半小时后小菜写出了第三版程序。

代码结构图



抽象状态类，定义一个抽象方法“写程序”

```
//抽象状态
public abstract class State
{
    public abstract void WriteProgram(Work w);
}
```

上午和中午工作状态类

```
//上午工作状态
public class ForenoonState : State
{
    public override void WriteProgram  (Work w)
    {
        if (w.Hour < 12)  {
            Console.WriteLine("当前时间：{0}点 上午工作，精神百倍", w.Hour);
        }
        else
        {
            w.SetState(new NoonState());w.WriteProgram();
        }
    }
}
```

超过 12 点，则转入中午工作状态

```
//中午工作状态
public class NoonState : State
{
    public override void WriteProgram(Work w)
    {
        if (w.Hour < 13)  {
            Console.WriteLine("当前时间：{0}点 饿了，午饭；犯困，午休。", w.Hour);
        }
        else
        {
            w.SetState(new AfternoonState());w.WriteProgram();
        }
    }
}
```

超过 13 点，则转入下午工作状态

下午和傍晚工作状态类

```
//下午工作状态
public class AfternoonState : State
{
    public override void WriteProgram  (Work w)
    {
     if (w.Hour < 17)
     {
         Console.WriteLine("当前时间：{0}点 下午状态还不错，继续努力", w.Hour);
     }
     else
     {
            w.SetState(new EveningState());
            w.WriteProgram();
     }
    }
}
```

超过 17 点，则转入傍晚工作状态

```
//晚间工作状态
public class EveningState : State
{
    public override void WriteProgram(Work w)
    {
        if (w.TaskFinished)
        {
            w.SetState(new RestState());
            w.WriteProgram();
        }
        else
        {
            if (w.Hour < 21)  {
               Console.WriteLine("当前时间：{0}点 加班哦，疲累之极", w.Hour);
            }
            else
            {
              w.SetState(new SleepingState());w.WriteProgram();
            }
        }
    }
}
```

如果完成任务，则转入下班状态

超过 21 点，则转入睡眠工作状态

睡眠状态和下班休息状态类

```
//睡眠状态
public class SleepingState : State
{
    public override void WriteProgram(Work w)
    {
        Console.WriteLine("当前时间：{0}点不行了，睡着了。",w.Hour);
    }
}
```

```
//下班休息状态
public class RestState : State
{
    public override void WriteProgram(Work w)
    {
        Console.WriteLine("当前时间：{0}点    下班回家了",w.Hour);
    }
}
```

工作类，此时没有了过长的分支判断语句。

```
//工作
public class Work
{
    private State current;
    public Work()
    {
        current = new ForenoonState();
    }

    private double hour;
    public double Hour
    {
        get { return hour; }
        set { hour = value; }
    }

    private bool finish = false;
    public bool TaskFinished
    {
        get { return finish; }
        set { finish = value; }
    }

    public void SetState(State s)
    {
        current = s;
    }

    public void WriteProgram()
    {
        current.WriteProgram(this);
    }
}
```

工作初始化为上午工作状态，即上午9点开始上班

“钟点”属性，状态转换的依据

“任务完成”属性，是否能下班的依据

客户端代码，没有任何改动。但我们的程序却更加灵活易变了。

```
static void Main(string[] args)
{
    //紧急项目
    Work emergencyProjects = new Work();
    emergencyProjects.Hour = 9;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 10;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 12;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 13;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 14;
    emergencyProjects.WriteProgram();
```

```
        emergencyProjects.Hour = 17;

        //emergencyProjects.WorkFinished = true;
        emergencyProjects.TaskFinished = false;

        emergencyProjects.WriteProgram();
        emergencyProjects.Hour = 19;
        emergencyProjects.WriteProgram();
        emergencyProjects.Hour = 22;
        emergencyProjects.WriteProgram();

        Console.Read();
    }
```

结果表现如下

```
当前时间：9点  上午工作，精神百倍
当前时间：10点  上午工作，精神百倍
当前时间：12点  饿了，午饭；犯困，午休
当前时间：13点  下午状态还不错，继续努力
当前时间：14点  下午状态还不错，继续努力
当前时间：17点  加班哦，疲累之极
当前时间：19点  加班哦，疲累之极
当前时间：22点  不行了，睡着了。
```

“此时的代码，如果要完成我所说的‘员工必须在20点之前离开公司’，我们只需要怎么样？”

“增加一个‘强制下班状态’，并改动一下‘傍晚工作状态’类的判断就可以了。而这是不影响其他状态的代码的。这样做的确是非常好。”

“哟，都半夜12点多了，快点睡觉吧。”大鸟提醒道。

“学会了状态模式，我的状态好着呢，让我再体会体会状态模式的美妙。”

“行了吧你，估计明上午的工作状态，就是睡觉打呼噜了。”

“嗨，这也是公司造成的呀。明天估计还得加班，无尽加班何时休，无尽加班何时休！”

第17章　在NBA我需要翻译——适配器模式

17.1　在NBA我需要翻译！

时间：4月22日 13点　　地点：小区外餐馆　　人物：小菜、大鸟

周日，小菜与大鸟上午在家刚看完NBA季后赛第一场比赛，出去吃饭时。

“大鸟，今天火箭开门红，赢得真是爽呀。”小菜感慨万分。

“是呀，希望能把这种势头保持到最后，那就可以有所突破了。”大鸟肯定说。

“你说姚明去了几年，英语也真练出来了哦，我看教练在那里布置战术，他旁边也没有翻译的，不住点头，瞧样子听懂没什么问题了。”

“要知道，在几年前，有记者问姚明说：‘在CBA和NBA最大的区别是什么？’，姚明的答案是‘在NBA我需要翻译，而在CBA我不需要。’经过四年的NBA锤炼，他的确是在NBA磨练中成长了。不但球技大涨，英语也学得非常棒，用英文答记者问一点问题都没有。不得不佩服呀。”

“钞票也大大地增加了，他可是中国最富有的体育明星。大鸟呀，你比他还大几岁吧，混得不行呀。”

“哪能和他比，两米二七的身高，你给我长一个试试。再说，单有身高也是不行的。在NBA现役中锋中，姚明也算是个天才吧。”

“你说当时他刚去美国时，怎么打球呀，什么都听不懂。”

“姚明爱说的一句名言叫‘皇帝不急急太监’，这种问题不是很容易吗，之前专门为他配备了翻译的，那个翻译一直在姚明身边，特别是比赛场上，教练、队员与他的对话全部都通过翻译来沟通。”

“想想看也是，如果不懂外语，又没有翻译，球技再高，估计也是不可能在国外待很长时间的。”

“哦，你等等，你的这个说法，倒让我想起一个设计模式，非常符合

你现在提到的这个场景。”

“是吗，听大鸟说设计模式已经成为习惯了，听不到都难受。快点说说看。”

17.2 适配器模式

“这个模式叫做适配器模式。”

> 适配器模式（**Adapter**），将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。[DP]

“适配器模式主要解决什么问题呢？”

“简单地说，就是需要的东西就在面前，但却不能使用，而短时间又无法改造它，于是我们就想办法适配它。”

“前面的听懂了，有东西不能用，又不能改造它。但想办法‘适配’是什么意思？”

“其实这个词应该是最早出现在电工学里，有些国家用110 V电压，而我们国家用的是220 V，但我们的电器，比如笔记本电脑是不能什么电压都能用的，但国家不同，电压可能不相同也是事实，于是就用一个电源适配器，只要是电，不管多少伏，都能把电源变成需要的电压，这就是电源适配器的作用。适配器的意思就是使得一个东西适合另一个东西的东西。”

“这个我明白，但和适配器模式有什么关系？”

“哈，NBA篮球运动员都会打篮球，姚明也会打篮球……”

“废话。”

“你小子，别打岔。但是姚明却不会英语，要在美国NBA打球，不会英语如何交流？没有交流如何理解教练和同伴的意图？又如何让他们理解自己的想法？不能沟通就打不好球了。于是就有三个办法，第一，让姚明学会英语，你看如何？”

“这不符合实际呀，姚明刚到NBA打球，之前又没有时间在学校里认真学好英语，马上学到可以听懂会说的地步是很困难的。”

“说得不错，第二种方法，让教练和球员学会中文？”

“哈，大鸟又在搞笑了。”

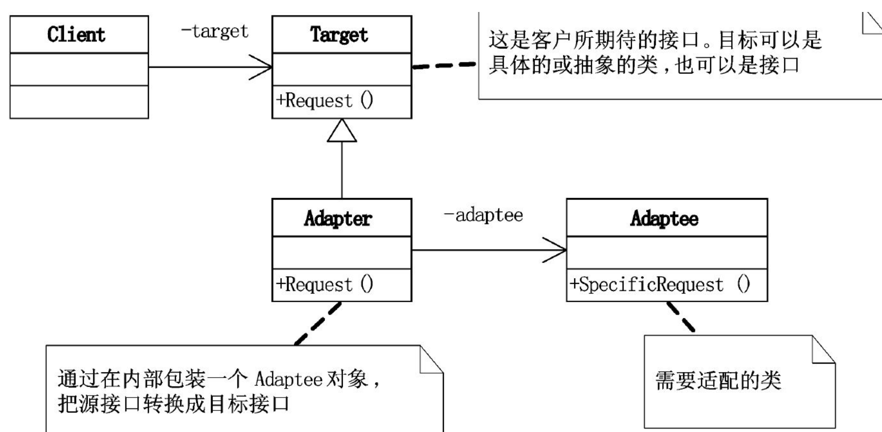
“不可能，那你说怎么办？”

“给姚明找个翻译。哦，我明白了，你的意思是翻译就是适配器？”

“对的，在我们不能更改球队的教练、球员和姚明的前题下，我们能做的就是想办法找个适配器。在软件开发中，也就是系统的数据和行为都正确，但接口不符时，我们应该考虑用适配器，目的是使控制范围之外的一个原有对象与某个接口匹配。适配器模式主要应用于希望复用一些现存的类，但是接口又与复用环境要求不一致的情况，比如在需要对早期代码复用一些功能等应用上很有实际价值。”

“在GoF的设计模式中，对适配器模式讲了两种类型，类适配器模式和对象适配器模式，由于类适配器模式通过多重继承对一个接口与另一个接口进行匹配，而C#、VB.NET、JAVA等语言都不支持多重继承（C++支持），也就是一个类只有一个父类，所以我们这里主要讲的是对象适配器。”

适配器模式（**Adapter**）结构图



Target（这是客户所期待的接口。目标可以是具体的或抽象的类，也可以是接口）代码如下。

```
class Target
{
    public virtual void Request()
    {
        Console.WriteLine("普通请求！");
    }
}
```

Adaptee（需要适配的类）代码如下：

```
class Adaptee
{
    public void SpecificRequest()
    {
        Console.WriteLine("特殊请求！");
    }
}
```

Adapter（通过在内部包装一个Adaptee对象，把源接口转换成目标接口）代码如下

```
class Adapter : Target
{
    private Adaptee adaptee = new Adaptee();

    public override void Request()
    {
        adaptee.SpecificRequest();
    }
}
```

建立一个私有的 Adaptee 对象

这样就可以把表面上调用 Request() 方法变成实际调用 SpecificRequest()

客户端代码如下：

```
static void Main(string[] args)
{
    Target target = new Adapter();
    target.Request();

    Console.Read();
}
```

对客户端来说，调用的就是 Target 的 Request()

17.3　何时使用适配器模式

“你的意思是不是说，在想使用一个已经存在的类，但如果它的接口，也就是它的方法和你的要求不相同时，就应该考虑用适配器模式？”

“对的，两个类所做的事情相同或相似，但是具有不同的接口时要使用它。而且由于类都共享同一个接口，使得客户代码如何？”

“客户代码可以统一调用同一接口就行了，这样应该可以更简单、更直接、更紧凑。”

“很好，其实用适配器模式也是无奈之举，很有点‘亡羊补牢’的感觉，没办法呀，是软件就有维护的一天，维护就有可能会因不同的开发人员、不同的产品、不同的厂家而造成功能类似而接口不同的情况，此时就是适配器模式大展拳脚的时候了。”

“你的意思是说，我们通常是在软件开发后期或维护期再考虑使用它？”

“如果是在设计阶段，你有必要把类似的功能类的接口设计得不同吗？”

“话是这么说，但不同的程序员定义方法的名称可能不同呀。”

“首先，公司内部，类和方法的命名应该有规范，最好前期就设计好，然后如果真的如你所说，接口不相同时，首先不应该考虑用适配器，而是应该考虑通过重构统一接口。”

“明白了，就是要在双方都不太容易修改的时候再使用适配器模式适配，而不是一有不同时就使用它。那有没有设计之初就需要考虑用适配器模式的时候？”

“当然有，比如公司设计一系统时考虑使用第三方开发组件，而这个组件的接口与我们自己的系统接口是不相同的，而我们也完全没有必要为了迎合它而改动自己的接口，此时尽管是在开发的设计阶段，也是可以考虑用适配器模式来解决接口不同的问题。”大鸟解释道，“好了，说了这么多，你都没有练习一下，来来来，试着把火箭队的比赛，教练叫暂停时给后卫、中锋、前锋分配进攻和防守任务的代码模拟出来。”

17.4 篮球翻译适配器

“哈，这有何难。后卫、中锋、前锋都是球员，所以应该有一个球员抽象类，有进攻和防守的方法。”

球员类

```
//球员
abstract class Player
{
    protected string name;
    public Player(string name)
    {
        this.name = name;
    }
    public abstract void Attack();
    public abstract void Defense();
}
```

进攻和防守方法

后卫、中锋、前锋类

```
//前锋
class Forwards : Player
{
    public Forwards(string name) : base(name)
    {
    }
    public override void Attack()
    {
        Console.WriteLine("前锋 {0} 进攻",name);
    }
    public override void Defense()
    {
        Console.WriteLine("前锋 {0} 防守",name);
    }
}

//中锋
class Center : Player
```

```
{
    //与前锋代码类似，略
}

//后卫
class Guards : Player
{
    //与前锋代码类似，略
}
```

客户端代码如下：

```
static void Main(string[] args)
{
    Player b = new Forwards("巴蒂尔");
    b.Attack();
    Player m = new Guards("麦克格雷迪");
    m.Attack();

    Player ym = new Center("姚明");
    ym.Attack();
    ym.Defense();

    Console.Read();
}
```

姚明问："Attack 和 Defense 是什么意思呢？"

结果显示

前锋 巴蒂尔 进攻
后卫 麦克格雷迪 进攻
中锋 姚明 进攻
中锋 姚明 防守

"注意，姚明刚来到NBA，他身高够高，球技够好，但是，他那时还不懂英语，也就是说，他听不懂教练的战术安排，Attack和Defense是什么意思不知道。你这样的写法就是有问题的。事实上，当时是如何解决这个矛盾的？"

“姚明说‘我需要翻译。’我知道你的意思了，姚明是外籍中锋，需要有翻译者类来‘适配’。”

外籍中锋

```
//外籍中锋
class ForeignCenter
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public void 进攻()
    {
        Console.WriteLine("外籍中锋 {0} 进攻", name);
    }

    public void 防守()
    {
        Console.WriteLine("外籍中锋 {0} 防守", name);
    }
}
```

外籍中锋类球员的姓名故意用属性而不是构造方法来区别与前三个球员类的不同

表明‘外籍中锋’只懂得中文‘进攻’

表明‘外籍中锋’只懂得中文‘防守’

翻译者类

```
//翻译者
class Translator : Player
{
    private ForeignCenter wjzf = new ForeignCenter();

    public Translator(string name)
        : base(name)
    {
        wjzf.Name = name;
    }

    public override void Attack()
    {
        wjzf.进攻();
    }

    public override void Defense()
    {
        wjzf.防守();
    }
}
```

声明并实例化一个内部‘外籍中锋’对象，表明翻译者与外籍球员有关联

翻译者将‘Attack’翻译为‘进攻’告诉外籍中锋

翻译者将‘Defense’翻译为‘防守’告诉外籍中锋

客户端代码改写如下：

```
static void Main(string[] args)
{
    Player b = new Forwards("巴蒂尔");
    b.Attack();

    Player m = new Guards("麦克格雷迪");
    m.Attack();

    Player ym = new Translator("姚明");
    ym.Attack();
    ym.Defense();

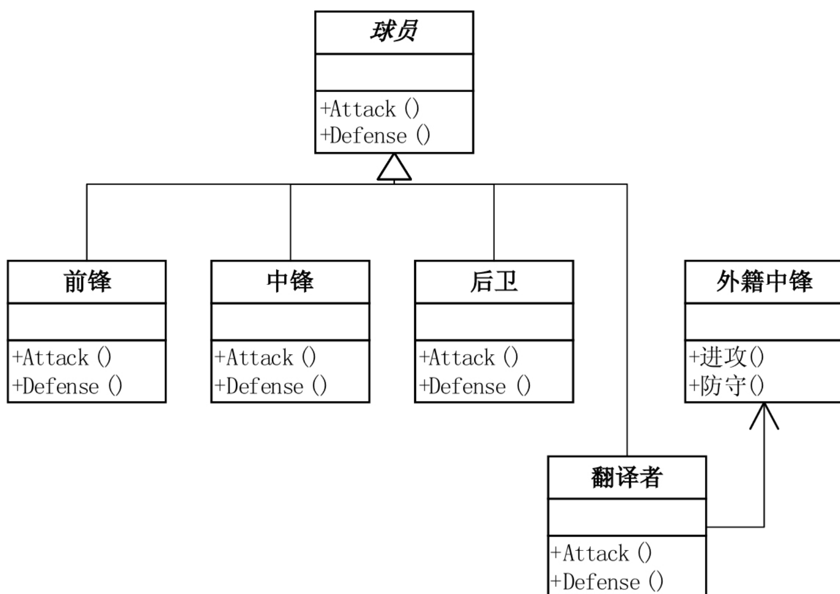
    Console.Read();
}
```

翻译者告诉姚明，教练要求你既要‘进攻’又要‘防守’

结果显示

前锋 巴蒂尔 进攻
后卫 麦克格雷迪 进攻
外籍中锋 姚明 进攻
外籍中锋 姚明 防守

代码结构图



“这下好了，尽管姚明曾经是不太懂英文，尽管火箭教练和球员也不会学中文，但因为有了翻译者，团队沟通合作成为了可能，非常好。”大

鸟鼓励道。

17.5　适配器模式的.NET应用

“这模式很好用，我想在现实中也很常用的吧。”

“当然，比如在.NET中有一个类库已经实现的、非常重要的适配器，那就是DataAdapter。**DataAdapter**用作**DataSet**和数据源之间的适配器以便检索和保存数据。**DataAdapter**通过映射**Fill**（这更改了**DataSet**中的数据以便与数据源中的数据相匹配）和**Update**（这更改了数据源中的数据以便与**DataSet**中的数据相匹配）来提供这一适配器**[MSDN]**。由于数据源可能是来自SQL　Server，可能来自Oracle，也可能来自Access、DB2，这些数据在组织上可能有不同之处，但我们希望得到统一的DataSet（实质是XML数据），此时用DataAdapter就是非常好的手段，我们不必关注不同数据库的数据细节，就可以灵活的使用数据。”

“啊，DataAdapter我都用了无数次了，原来它就是适配器模式的应用呀，太棒了。我喜欢这个模式，我要经常性地使用它。”

“NO、NO、NO！模式乱用不如不用。我给你讲个小故事吧，希望你能理解其深意。”

17.6　扁鹊的医术

大鸟："当年，魏文王问名医扁鹊说：'你们家兄弟三人，都精于医术，到底哪一位最好呢？'扁鹊答：'长兄最好，中兄次之，我最差。'文王再问：'那么为什么你最出名呢？'扁鹊答：'长兄治病，是治病于病情发作之前。由于一般人不知道他事先能铲除病因，所以他的名气无法传出去；中兄治病，是治病于病情初起时。一般人以为他只能治轻微的小病，所以他的名气只及本乡里。而我是治病于病情严重之时。一般人都看到我在经脉上穿针管放血、在皮肤上敷药等大手术，所以大家都以为我的医术高明，名气因此响遍全国。'这个故事说明什么？"

"啊，你的意思是如果能事先预防接口不同的问题，不匹配问题就不会发生；在有小的接口不统一问题发生时，及时重构，问题不至于扩大；只有碰到无法改变原有设计和代码的情况时，才考虑适配。事后控制不如事中控制，事中控制不如事前控制。"小菜总结说。

"对呀，如果能事前控制，又何必要事后再去弥补呢？"大鸟肯定说，"适配器模式当然是好模式，但如果无视它的应用场合而盲目使用，其实是本末倒置了。"

"嗯，我相信，小菜我能把适配器模式应用到扁鹊的境界的。哦，不对，是他们三兄弟共同的境界。我一定能！"

"相信你一定能。"

第18章　如果再回到从前——备忘录模式

18.1　如果再给我一次机会……

时间：5月6日18点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“小菜，今天上午看NBA了吗？火箭季后赛第七场对爵士的比赛。”大鸟问道。

“没有，不过结果倒是在网上第一时间就知道了。在最后比赛剩下4分钟时，比分还相同，到剩下57.8秒的时候，火箭也只落后2分，可惜，最后的两个进攻篮板球没得到，火箭就输掉了比赛。”

“是呀，最后一分钟的失误，几乎就等于输掉了整个赛季。”

“如果火箭任何一人能抓到两个篮板中的一个，结果可能完全不是这样。真是遗憾呀。”小菜感慨道。

“很多时候我们做了件事后，却开始后悔。这就是人类的内心软弱一面。时间不能倒流，不管怎么样人生是无法回到从前的，但是软件就不一样了。还记得玩一些单机的PC游戏的时候吗，通常我都是在打大Boss之前，先保存一个进度，然后如果通关失败了，我可以再返回刚才那个进度来恢复原来的状态，从头来过。从这点上说，我们比姚明强。”

“哈，这其中原理是不是就是把当前的游戏状态的各种参数存储，以便恢复时读取呢？”

“是的，通常这种保存都是存在磁盘上了，以便日后读取。但对于一些更为常规的应用，比如我们下棋时需要悔棋、编写文档时需要撤销、查看网页时需要后退，这些相对频繁而简单的恢复并不需要存在磁盘中，只要将保存在内存中的状态恢复一下即可。”

“嗯，这是更普通的应用，很多开发中都会用到。”

“那我简单说个场景，你想想看怎么用代码实现。游戏的某个场景，一游戏角色有生命力、攻击力、防御力等等数据，在打Boss前和后一定会不一样的，我们允许玩家如果感觉与Boss决斗的效果不理想可以让游

戏恢复到决斗前。”

“好的，我试试看。”

18.2 游戏存进度

游戏角色类，用来存储角色的生命力、攻击力、防御力的数据。

```
class GameRole
{
    //生命力
    private int vit;
    public int Vitality
    {
        get { return vit; }  set { vit = value; }
```

```
    }
    //攻击力
    private int atk;
    public int Attack
    {
        get { return atk; }  set { atk = value; }
    }
    //防御力
    private int def;
    public int Defense
    {
        get { return def; }  set { def = value; }
    }
    //状态显示
    public void StateDisplay()
    {
        Console.WriteLine("角色当前状态：");
        Console.WriteLine("体力：{0}", this.vit);
        Console.WriteLine("攻击力：{0}", this.atk);
        Console.WriteLine("防御力：{0}", this.def);
        Console.WriteLine("");
    }
    //获得初始状态
    public void GetInitState()
    {
        this.vit = 100;
        this.atk = 100;
        this.def = 100;
    }
    //战斗
    public void Fight()
    {
        this.vit = 0;
        this.atk = 0;
        this.def = 0;
    }
}
```

数据通常来自本机磁盘或远程数据库

在与 Boss 大战后游戏数据损耗为零

客户端调用时

```
static void Main(string[] args)
{
    //大战 Boss 前
    GameRole lixiaoyao = new GameRole();
    lixiaoyao.GetInitState();
    lixiaoyao.StateDisplay();

    //保存进度
    GameRole backup = new GameRole();
    backup.Vitality = lixiaoyao.Vitality;
    backup.Attack = lixiaoyao.Attack;
    backup.Defense = lixiaoyao.Defense;

    //大战 Boss 时，损耗严重
    lixiaoyao.Fight();
    lixiaoyao.StateDisplay();

    //恢复之前状态
    lixiaoyao.Vitality = backup.Vitality;
    lixiaoyao.Attack = backup.Attack;
    lixiaoyao.Defense = backup.Defense;

    lixiaoyao.StateDisplay();

    Console.Read();
}
```

大战 Boss 前，获得初始角色状态

通过‘游戏角色’的新实例，来保存进度

大战 Boss 时，损耗严重所有数据全部损耗为零

GameOver 不甘心，恢复之前进度，重新来玩

结果显示

角色当前状态：
体力：100
攻击力：100
防御力：100

角色当前状态：
体力：0
攻击力：0
防御力：0

角色当前状态：
体力：100
攻击力：100
防御力：100

“小菜，这样的写法，确实是实现了我的要求，但是问题也确实多

多。”

“哈，你的经典理论，代码无错未必优。说吧，我有心理准备。”

“问题主要在于这客户端的调用。下面这一段有问题，因为这样写就把整个游戏角色的细节暴露给了客户端，你的客户端的职责就太大了，需要知道游戏角色的生命力、攻击力、防御力这些细节，还要对它进行‘备份’。以后需要增加新的数据，例如增加‘魔法力’或修改现有的某种力，例如‘生命力’改为‘经验值’，这部分就一定要修改了。同样的道理也存在于恢复时的代码。”

```
GameRole backup = new GameRole();
backup.Vitality = lixiaoyao.Vitality;
backup.Attack = lixiaoyao.Attack;
backup.Defense = lixiaoyao.Defense;
```

暴露了实现细节，不足取

```
lixiaoyao.Vitality = backup.Vitality;
lixiaoyao.Attack = backup.Attack;
lixiaoyao.Defense = backup.Defense;
```

同样暴露了实现细节，不足取

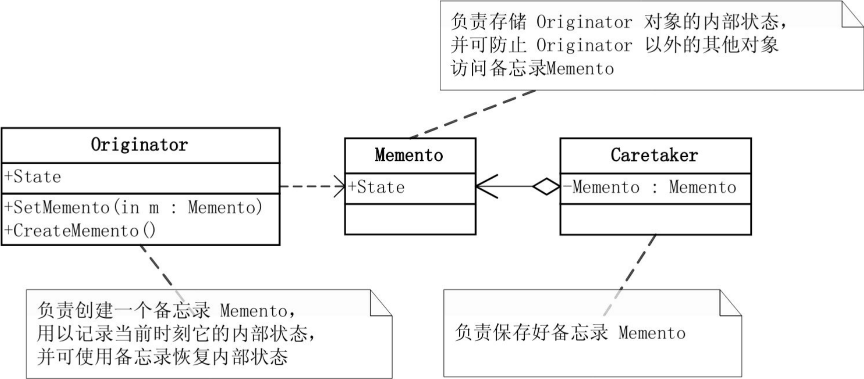
“显然，我们希望的是把这些‘游戏角色’的存取状态细节封装起来，而且最好是封装在外部的类当中。以体现职责分离。”

18.3 备忘录模式

“所以我们需要学习一个新的设计模式，备忘录模式。”

> **备忘录（Memento）**：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。[DP]

备忘录模式（**Memento**）结构图



Originator（发起人）：负责创建一个备忘录Memento，用以记录当前时刻它的内部状态，并可使用备忘录恢复内部状态。Originator可根据需要决定Memento存储Originator的哪些内部状态。

Memento（备忘录）：负责存储Originator对象的内部状态，并可防止Originator以外的其他对象访问备忘录Memento。备忘录有两个接口，Caretaker只能看到备忘录的窄接口，它只能将备忘录传递给其他对象。Originator能够看到一个宽接口，允许它访问返回到先前状态所需的所有数据。

Caretaker（管理者）：负责保存好备忘录Memento，不能对备忘录的内容进行操作或检查。

“就刚才的例子，‘游戏角色’类其实就是一个Originator，而你用了同样的‘游戏角色’实例‘备份’来做备忘录，这在当需要保存全部信息时，是可以考虑的，而用clone的方式来实现Memento的状态保存可能是更好的办法，但是如果是这样的话，使得我们相当于对上层应用开放了Originator的全部（public）接口，这对于保存备份有时候是不合适的。”

“那如果我们不需要保存全部的信息以备使用时，怎么办？”

“哈，对的，这或许是更多可能发生的情况，我们需要保存的并不是全部信息，而只是部分，那么就应该有一个独立的备忘录类Memento，它只拥有需要保存的信息的属性。”

18.4 备忘录模式基本代码

发起人（Originator）类

```
class Originator
{
    private string state;
    public string State
    {
        get { return state; }
        set { state = value; }
    }
    public Memento CreateMemento()
    {
        return (new Memento(state));
    }
    public void SetMemento(Memento memento)
    {
        state = memento.State;
    }
    public void Show()
    {   Console.WriteLine("State=" + state);    }
}
```

需要保存的属性，可能有多个

创建备忘录，将当前需要保存的信息导入并实例化出一个 Memento 对象

恢复备忘录，将 Memento 导入并将相关数据恢复

显示数据

备忘录（Memento）类

```
class Memento
{
    private string state;

    public Memento(string state)
    {
        this.state = state;
    }

    public string State
    {
        get { return state; }
    }
}
```

构造方法，将相关数据导入

需要保存的数据属性，可以是多个

管理者（Caretaker）类

```
class Caretaker
{
    private Memento memento;

    public Memento Memento          // 得到或设置备忘录
    {
        get { return memento; }
        set { memento = value; }
    }
}
```

客户端程序

```
static void Main(string[] args)
{

    Originator o = new Originator();     // Originator 初始状态，状态属性为“On”
    o.State = "On";
    o.Show();

    Caretaker c = new Caretaker();
    c.Memento = o.CreateMemento();       // 保存状态时，由于有了很好的封装，可以隐藏 Originator 的实现细节

    o.State = "Off";
    o.Show();                            // Originator 改变了状态属性为“Off”

    o.SetMemento(c.Memento);
    o.Show();                            // 恢复原初始状态

    Console.Read();
}
```

“哈，我明白了，这当中就是把要保存的细节给封装在了**Memento**中了，哪一天要更改保存的细节也不用影响客户端了。那么这个备忘录模式都用在一些什么场合呢？”

“**Memento**模式比较适用于功能比较复杂的，但需要维护或记录属性历史的类，或者需要保存的属性只是众多属性中的一小部分时，**Originator**可以根据保存的**Memento**信息还原到前一状态。”

“我记得好像命令模式也有实现类似撤销的作用？”

“哈，小子记性不错，如果在某个系统中使用命令模式时，需要实现命令的撤销功能，那么命令模式可以使用备忘录模式来存储可撤销操作的状态 **[DP]**。有时一些对象的内部信息必须保存在对象以外的地方，但是必须要由对象自己读取，这时，使用备忘录可以把复杂的对象内部信息对其他的对象屏蔽起来 **[DP]** ，从而可以恰当地保持封装的边界。”

“我感觉可能最大的作用还是在当角色的状态改变的时候，有可能这个状态无效，这时候就可以使用暂时存储起来的备忘录将状态复原 **[DP]** 这个作用吧？”

“说得好，这当然是最重要的作用了。”

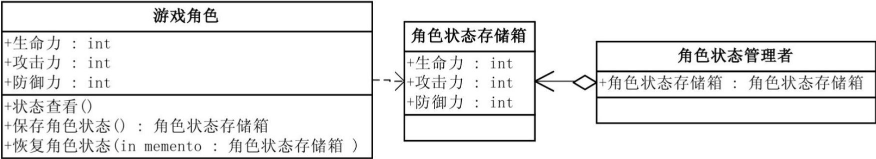
“明白，我学会了。”

“别急，你还没有把你刚才的代码改成备忘录模式的。”

“啊，你就不打算饶过我。等着，看我来拿满分。”

18.5 游戏进度备忘

代码结构图



游戏角色类

```
class 游戏角色
{
    ............

    //保存角色状态
    public RoleStateMemento SaveState()
    {
        return (new RoleStateMemento(vit, atk, def));
    }

    //恢复角色状态
    public void RecoveryState(RoleStateMemento memento)
    {
        this.vit = memento.Vitality;
        this.atk = memento.Attack;
        this.def = memento.Defense;
    }
    ............

}
```

新增“保存角色状态”方法，将游戏角色的三个状态值通过实例化“角色状态存储箱”返回

新增“恢复角色状态”方法，可将外部的“角色状态存储箱”中状态值恢复给游戏角色

角色状态存储箱类

```
//角色状态存储箱
class RoleStateMemento
{
    private int vit;
    private int atk;
    private int def;
    public RoleStateMemento(int vit, int atk, int def)
    {
        this.vit = vit;
        this.atk = atk;
        this.def = def;
    }
    //生命力
    public int Vitality
    {
        get { return vit; }
        set { vit = value; }
    }
    //攻击力
    public int Attack
    {
        get { return atk; }
        set { atk = value; }
    }
    //防御力
    public int Defense
    {
        get { return def; }
        set { def = value; }
    }
}
```

将生命力、攻击力、防御力存入状态存储箱对象中

角色状态管理者类

```
//角色状态管理者
class RoleStateCaretaker
{
    private RoleStateMemento memento;

    public RoleStateMemento Memento
    {
        get { return memento; }
        set { memento = value; }
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    //大战 Boss 前
    GameRole lixiaoyao = new GameRole();
    lixiaoyao.GetInitState();
    lixiaoyao.StateDisplay();

    //保存进度
    RoleStateCaretaker stateAdmin = new RoleStateCaretaker();
    stateAdmin.Memento = lixiaoyao.SaveState();

    //大战 Boss 时，损耗严重
    lixiaoyao.Fight();
    lixiaoyao.StateDisplay();

    //恢复之前状态
    lixiaoyao.RecoveryState(stateAdmin.Memento);
    lixiaoyao.StateDisplay();

    Console.Read();
}
```


“看看，能不能得满分，我查了好几遍了。”

“不错，写得还行。你要注意，备忘录模式也是有缺点的，角色状态需要完整存储到备忘录对象中，如果状态数据很大很多，那么在资源消耗上，备忘录对象会非常耗内存。”

“嗯，明白。所以也不是用得越多越好。”

“小子，以后打游戏要记着用备忘录哦。”大鸟不忘提醒一句。

“哈，我一定会这样。”小菜开始装着深沉地说，“曾经有一个精彩的游戏摆在我的面前，但是我没有好好珍惜。等到死于Boss手下的时候才后悔莫及，尘世间最痛苦的事莫过于此。如果上天可以给我一个机会再来一次的话，我会对你说三个字，‘存进度’。如果非要把这个进度加上一保险，我希望是刻成光盘，流传万年！”

第19章　分公司＝一部门——组合模式

19.1　分公司不就是一部门吗？

时间：5月10日19点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“大鸟，请教你一个问题？快点帮帮我。”

“今天轮到你做饭，你可别忘记了。”

“做饭？好说好说！先帮我解决问题吧。再弄不出来，我要失业了。”

“有这么严重吗？什么问题呀。”

“我们公司最近接了一个项目，是为一家在全国许多城市都有分销机构的大公司做办公管理系统，总部有人力资源、财务、运营等部门。”

“这是很常见的OA系统，需求分析好的话，应该不难开发的。”

“是呀，我开始也这么想，这家公司试用了我们开发的系统后感觉不错，他们希望可以在他们的全部分公司推广，一起使用。他们在北京有总部，在全国几大城市设有分公司，比如上海设有华东区分部，然后在一些省会城市还设有办事处，比如南京办事处、杭州办事处。现在有个问题是，总公司的人力资源部、财务部等办公管理功能在所有的分公司或办事处都需要有。你说怎么办？”

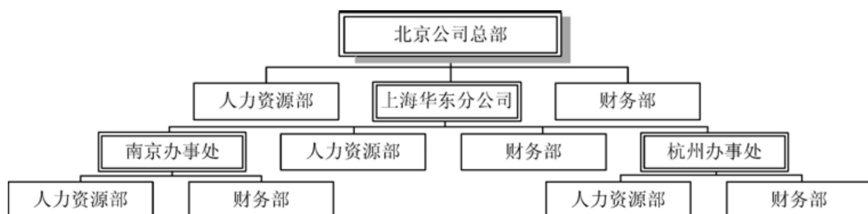
“你打算怎么办呢？”大鸟不答反问道。

“因为你之前讲过简单复制是最糟糕的设计，所以我的想法是共享功能到各个分公司，也就是让总部、分公司、办事处用同一套代码，只是根据ID的不同来区分。”

“要糟了。”

“你怎么知道，的确是不行，因为他们的要求，总部、分部和办事处是成树状结构的，也就是有组织结构的，不可以简单的平行管理。这下

我就比较痛苦了，因为实际开发时就得一个一个的判断它是总部，还是分公司的财务，然后再执行其相应的方法。”



“你有没有发现，类似的这种部分与整体情况很多见，例如卖电脑的商家，可以卖单独配件也可以卖组装整机，又如复制文件，可以一个一个文件复制粘贴还可以整个文件夹进行复制，再比如文本编辑，可以给单个字加粗、变色、改字体，当然也可以给整段文字做同样的操作。其本质都是同样的问题。”

“你的意思是，分公司或办事处与总公司的关系，就是部分与整体的关系？”

“对的，你希望总公司的组织结构，比如人力资源部、财务部的管理功能可以复用于分公司。这其实就是整体与部分可以被一致对待的问题。”

“哈，我明白了，就像你举的例子，对于Word文档里的文字，对单个字的处理和对多个字、甚至整个文档的处理，其实是一样的，用户希望一致对待，程序开发者也希望一致处理。但具体怎么做呢？”

“首先，我们来分析一下你刚才讲到的这个项目，如果把北京总公司当做一棵大树的根部的话，它的下属分公司其实就是这棵树的什么？”

“是树的分枝，哦，至于各办事处是更小的分支，而它们的相关的职能部门由于没有分枝了，所以可以理解为树叶。”

“小菜理解得很快，尽管天下没有两片相同的树叶，但同一棵树上长出来的树叶样子也不会相差到哪去。也就是说，你所希望的总部的财务部管理功能也最好是能复用到子公司，那么最好的办法就是，我们在处理总公司的财务管理功能和处理子公司的财务管理功能的方法都是一样的。”

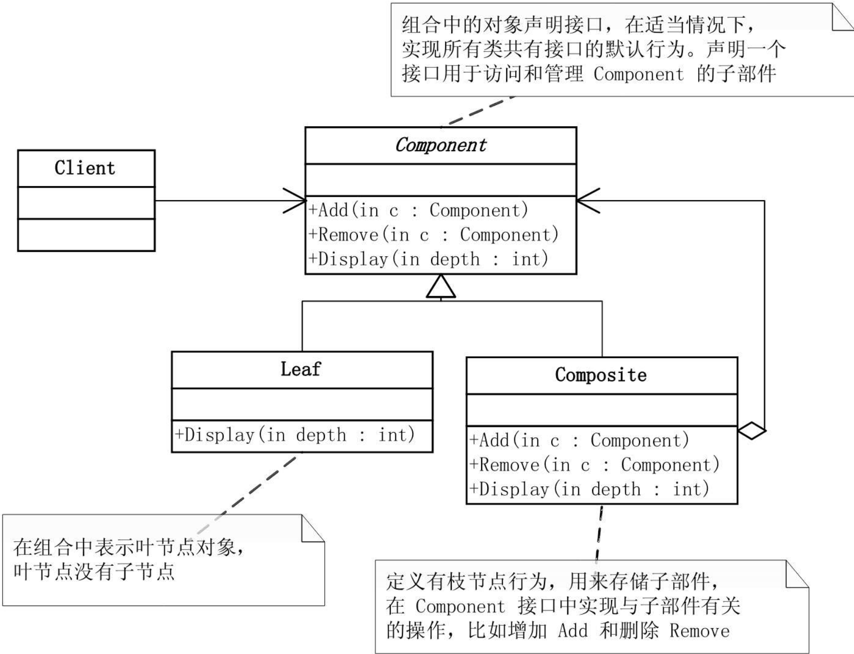
“有点晕了，别绕弯子了，你是不是想讲一个新的设计模式给我。”

19.2　组合模式

“哈，小菜够直接。这个设计模式叫做‘组合模式’。”

> 组合模式（**Composite**），将对象组合成树形结构以表示‘部分-整体’的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。[DP]

组合模式（Composite）结构图



Component为组合中的对象声明接口，在适当情况下，实现所有类共有接口的默认行为。声明一个接口用于访问和管理Component的子部件。

```
abstract class Component
{
    protected string name;

    public Component(string name)
    {
        this.name = name;
    }

    public abstract void Add(Component c);
    public abstract void Remove(Component c);
    public abstract void Display(int depth);
}
```

通常都用 Add 和 Remove 方法来提供增加或移除树叶或树枝的功能

Leaf在组合中表示叶节点对象，叶节点没有子节点。

```
class Leaf : Component
{
    public Leaf(string name)
        : base(name)
    { }

    public override void Add(Component c)
    {
        Console.WriteLine("Cannot add to a leaf");
    }

    public override void Remove(Component c)
    {
        Console.WriteLine("Cannot remove from a leaf");
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
    }
}
```

由于叶子没有再增加分枝和树叶，所以 Add 和 Remove 方法实现它没有意义，但这样做可以消除叶节点和枝节点对象在抽象层次的区别，它们具备完全一致的接口

叶节点的具体方法，此处是显示其名称和级别

Composite定义有枝节点行为，用来存储子部件，在Component接口中实现与子部件有关的操作，比如增加Add和删除Remove。

```
class Composite : Component
{
    private List<Component>children=new List<Component>();

    public Composite(string name)
        : base(name)
    { }

    public override void Add(Component c)
    {
        children.Add(c);
    }

    public override void Remove(Component c)
    {
        children.Remove(c);
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);

        foreach (Component component in children)
        {
            component.Display(depth + 2);
        }
    }
}
```

一个子对象集合用来存储其下属的枝节点和叶节点

显示其枝节点名称，并对其下级进行遍历

客户端代码，能通过Component接口操作组合部件的对象。

```
static void Main(string[] args)
{
    Composite root = new Composite("root");
    root.Add(new Leaf("Leaf A"));
    root.Add(new Leaf("Leaf B"));

    Composite comp = new Composite("Composite X");
    comp.Add(new Leaf("Leaf XA"));
    comp.Add(new Leaf("Leaf XB"));

    root.Add(comp);

    Composite comp2 = new Composite("Composite XY");
    comp2.Add(new Leaf("Leaf XYA"));
    comp2.Add(new Leaf("Leaf XYB"));
    comp.Add(comp2);

    root.Add(new Leaf("Leaf C"));

    Leaf leaf = new Leaf("Leaf D");
    root.Add(leaf);
    root.Remove(leaf);

    root.Display(1);

    Console.Read();
}
```

生成树根 root，根上长出两叶 LeafA 和 LeafB

根上长出分枝 Composite X，分枝上也有两叶 LeafXA 和 LeafXB

在 Composite X 上再长出分枝 Composite XY，分枝上也有两叶 LeafXYA 和 LeafXYB

根部又长出两叶 LeafC 和 LeafD，可惜 LeafD 没长牢，被风吹走了

显示大树的样子

结果显示

```
-root
---Leaf A
---Leaf B
---Composite X
-----Leaf XA
-----Leaf XB
-----Composite XY
-------Leaf XYA
-------Leaf XYB
---Leaf C
```

19.3　透明方式与安全方式

“树可能有无数的分枝，但只需要反复用Composite就可以实现树状结构了。小菜感觉如何？”

“有点懂，但还是有点疑问，为什么Leaf类当中也有Add和Remove，树叶不是不可以再长分枝吗？”

“是的，这种方式叫做透明方式，也就是说在**Component**中声明所有用来管理子对象的方法，其中包括**Add**、**Remove**等。这样实现**Component**接口的所有子类都具备了**Add**和**Remove**。这样做的好处就是叶节点和枝节点对于外界没有区别，它们具备完全一致的行为接口。但问题也很明显，因为**Leaf**类本身不具备**Add**（）、**Remove**（）方法的功能，所以实现它是没有意义的。”

“哦，那么如果我不希望做这样的无用功呢？也就是Leaf类当中不用Add和Remove方法，可以吗？”

“当然是可以，那么就需要安全方式，也就是在**Component**接口中不去声明**Add**和**Remove**方法，那么子类的**Leaf**也就不需要去实现它，而是在**Composite**声明所有用来管理子类对象的方法，这样做就不会出现刚才提到的问题，不过由于不够透明，所以树叶和树枝类将不具有相同的接口，客户端的调用需要做相应的判断，带来了不便。”

“那我喜欢透明式，那样就不用做任何判断了。”

“开发怎么能随便有倾向性？两者各有好处，视情况而定吧。”

19.4 何时使用组合模式

“什么地方用组合模式比较好呢？”

“当你发现需求中是体现部分与整体层次的结构时，以及你希望用户可以忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象时，就应该考虑用组合模式了。”

“哦，我想起来了。以前曾经用过的ASP.NET的TreeView控件就是典型的组合模式应用。”

“又何止是这个，你应该写过自定义控件吧，也就是把一些基本的控件组合起来，通过编程写成一个定制的控件，比如用两个文本框和一个按钮就可以写一下自定义的登录框控件，实际上，所有的Web控件的基类都是System.Web.UI.Control，而Control基类中就有Add和Remove方法，这就是典型的组合模式的应用。”

“哦，对的对的，这就是部分与整体的关系。”

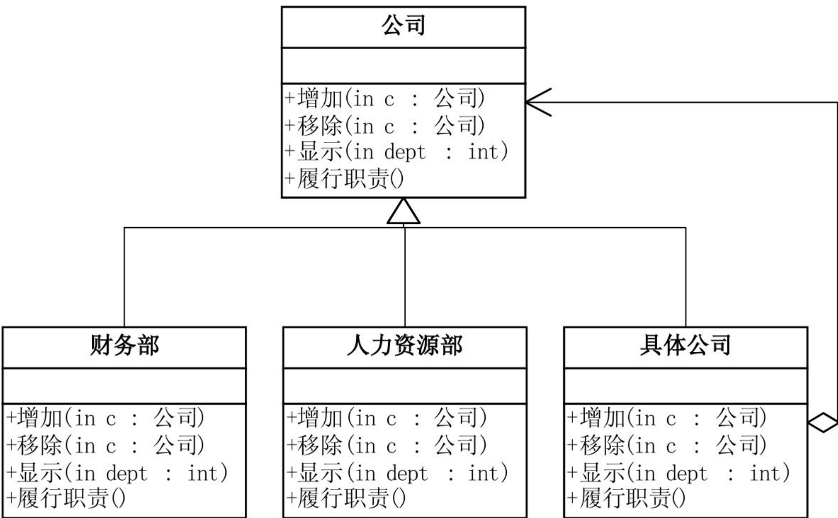
“好了，你是不是可以把你提到的公司管理系统的例子练习一下了？”

“OK，现在感觉不是很困难了。”

19.5 公司管理系统

半小时后，小菜写出了代码。

代码结构图



公司类 抽象类或接口

```
abstract class Company
{
    protected string name;

    public Company(string name)
    {
        this.name = name;
    }

    public abstract void Add(Company c);//增加
    public abstract void Remove(Company c);//移除
    public abstract void Display(int depth);//显示
    public abstract void LineOfDuty();//履行职责
}
```

增加一“履行职责”方法，不同的部门需履行不同的职责

具体公司类 实现接口 树枝节点

```
class ConcreteCompany : Company
{
```

```
    private List<Company>children=new List<Company>()

    public ConcreteCompany(string name)
        : base(name)
    { }

    public override void Add(Company c)
    {
        children.Add(c);
    }

    public override void Remove(Company c)
    {
        children.Remove(c);
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-',
depth) + name);

        foreach(Company component in children)
        {
            component.Display(depth + 2);
        }
    }

    //履行职责

    public override void LineOfDuty()
    {
        foreach(Company component in children)
        {
            component.LineOfDuty();
        }
    }
}
```

人力资源部与财务部类 树叶节点

```
//人力资源部
```

```
class HRDepartment : Company
{
    public HRDepartment(string name) : base(name)
    { }

    public override void Add(Company c)
    { }

    public override void Remove(Company c)
    { }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-',
depth) + name);
    }

    public override void LineOfDuty()
    {
        Console.WriteLine("{0} 员工招聘培训管理",
name);
    }
}

//财务部
class FinanceDepartment : Company
{
    public FinanceDepartment(string name) : base(na
    { }

    public override void Add(Company c)
    { }

   public override void Remove(Company c)
   { }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-',
depth) + name);
    }

    public override void LineOfDuty()
    {
```

```
        Console.WriteLine("{0} 公司财务收支管理",
name);
    }
}
```

客户端调用

```
static void Main(string[] args)
{
    ConcreteCompany root = new ConcreteCompany("北
京总公司");
    root.Add(new HRDepartment("总公司人力资源
部"));
    root.Add(new FinanceDepartment("总公司财务
部"));

    ConcreteCompany comp = new ConcreteCompany("上
海华东分公司");
    comp.Add(new HRDepartment("华东分公司人力资源
部"));
    comp.Add(new FinanceDepartment("华东分公司财务
部"));
    root.Add(comp);

    ConcreteCompany comp1 = new ConcreteCompany("南
京办事处");
    comp1.Add(new HRDepartment("南京办事处人力资源
部"));
    comp1.Add(new FinanceDepartment("南京办事处财
务部"));
    comp.Add(comp1);

    ConcreteCompany comp2 = new ConcreteCompany("杭
州办事处");
    comp2.Add(new HRDepartment("杭州办事处人力资源
部"));
    comp2.Add(new FinanceDepartment("杭州办事处财
务部"));
    comp.Add(comp2);

    Console.WriteLine("\n结构图：");
```

```
    root.Display(1);

    Console.WriteLine("\n职责：");
    root.LineOfDuty();

    Console.Read();
}
```

结果显示

```
结构图：
-北京总公司
---总公司人力资源部
---总公司财务部
---上海华东分公司
-----华东分公司人力资源部
-----华东分公司财务部
-----南京办事处
-------南京办事处人力资源部
-------南京办事处财务部
-----杭州办事处
-------杭州办事处人力资源部
-------杭州办事处财务部

职责：
总公司人力资源部 员工招聘培训管理
总公司财务部 公司财务收支管理
华东分公司人力资源部 员工招聘培训管理
华东分公司财务部 公司财务收支管理
南京办事处人力资源部 员工招聘培训管理
南京办事处财务部 公司财务收支管理
杭州办事处人力资源部 员工招聘培训管理
杭州办事处财务部 公司财务收支管理
```

19.6　组合模式好处

“小菜写得不错，你想想看，这样写的好处有哪些？”

“组合模式这样就定义了包含人力资源部和财务部这些基本对象和分公司、办事处等组合对象的类层次结构。基本对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合，这样不断地递归下去，客户代码中，任何用到基本对象的地方都可以使用组合对象了。”

“非常好，还有没有？”

“我感觉用户是不用关心到底是处理一个叶节点还是处理一个组合组件，也就用不着为定义组合而写一些选择判断语句了。”

“简单点说，就是组合模式让客户可以一致地使用组合结构和单个对象。”

“这也就是说，那家公司开多少个以及多少级办事处都没问题了。”小菜开始兴奋起来同，“哪怕开到地级市、县级市、镇、乡、村、户……”

“喂，发什么神经了。”大鸟提醒道，“开办事处到户？你有毛病呀。”

“不过理论上，用了组合模式，在每家每户设置一个人力资源部和财务部也是很正常的。”小菜得意地说，“哪家不需要婚丧嫁娶、增丁添口等家务事，哪家不需要柴米油盐、衣食住行等流水账。”

“你小子，刚才还在为项目设计不好而犯愁叫失业，现在可好，得意得恨不得全国挨家挨户用你那套软件，瞧你那德行。”

“我就这德行，学到东西，水平当然就不同了。我去考虑真实的设计了。”

“小菜，今天该轮到你烧饭做菜，别想逃。”

“不是还有点剩饭吗？”

“没有菜如何吃呀。”

“大鸟呀，用组合模式呀，如你所说，客户是不用关心吃什么，直接吃米饭或者吃饭菜组合，其效果对客户来说都是填饱肚子，你就将就一下吧。”小菜说完，就逃离了大鸟房间。

“啊，有这样应用组合模式的？你给我回来。”大鸟叫道。

第20章　想走？可以！先买票——迭代器模式

20.1　乘车买票，不管你是谁！

时间：5月26日10点　　地点：一部公交车上　　人物：小菜、大鸟、售票员、公交乘客、小偷

这天是周末，小菜和大鸟一早出门游玩，上了公交车，车内很拥挤。

“上车的乘客请买票。”售票员一边在人缝中穿插着，一边说道。

……

“先生，您这包行李太大了，需要补票的。”售票员对一位拿着大包行李的乘客说道。

“哦，这也需要买票呀，它又不是人。”带大包行李的乘客说。

“您可以看看规定，当您携带的行李占据了一个客用面积时，请再购买同程车票一张，谢谢合作。”售票员指了指车上的一个纸牌子。

这位乘客很不情愿地再买了一张票。

“还有三位乘客没有买票，请买票！”

……

“这售票员够厉害，记得这么清楚，上来几个人都记得。”小菜感叹道。

“这也是业务能力的体现呀。”大鸟解释说。

……

“先生请买票！”售票员对着一位老外说道。

“Sorry，What do you say?”老外看来不会中文。

“请买车票怎么说？”售票员低声的自言自语道，“Please buy ……票怎么说……”

“ticket.”小菜手掌放嘴边，小声地提醒了一句。

“谢谢，”售票员对小菜笑了笑，接着用中国式英文对着老外说道，“Please buy ticket.”

“Oh！yes.”老外急忙掏钱包拿了一张十元人民币。

“买票了，买票了，还有两位，不要给不买票的人任何机会……”售票员找了老外钱后吆喝着，又对着一穿着同样公交制服的女的说道，“小姐，请买票！”

“我也是公交公司的，”这女的拿出一个公交证件，在售票员面前晃了晃。

“不好意思，公司早就出规定了，工作证不得作为乘车凭证。”售票员说道。

“我乘车从来就没买过票，凭什么在这就要买票。”这个乘客开始耍赖。

此时旁边的乘客都来劲了，七嘴八舌说起来。

“公交公司的员工就不是乘客呀，国家总理来也要买票的。”

“这人怎么这样，想占大伙的便宜呀。”

“你还当过去呀，现在二十一世纪不吃大锅饭了。欠债还钱，乘车买票，天经地义……”

“行了行了，不就是一张票吗，搞什么搞，呐！买票。”这不想买票的小姐终于扛不住了，递出去两元钱买了票。

“还有哪一位没有买票，请买票。”售票员继续在拥挤的车厢里跋涉着。

“小偷！你这小偷，把手机还我。”突然站在小菜不远处的一小姑娘对着一猥琐的男人叫了起来。

“你不要乱讲，我哪有偷你手机。”

“我看见你刚才把手伸进了我的包里。就是你偷的。”

“我没有偷，你看错了。”

“我明明看见你偷的。”小姑娘急得哭了出来。

小菜看不过去了，“你的手机号多少，我帮你打打看。”

“138xxxx8888”小姑娘像是看到了希望。

“哇，这么强的号，手机一定不会丢。”小菜羡慕着，用自己的手机拨了这个号码。

那人眼看着不对，想往门口跑，小菜和大鸟冲了上去，一把按住他。

“你看，我的手机响了，就在他身上。”小姑娘叫了起来，“就是他，他就是小偷。”

此时两个小伙已经把猥琐男死死按在了地板上。

“快打110报警！”大鸟喊道。

此时公交车也停了下来，所有的乘客都议论着“小偷真可恶”的话题。

不一会，民警来了，问清楚了来由，正准备将小偷带走时，售票员对着小偷发话了：“慢着，你是那个没有买票的人吧？”

“啊？嗯！是的。”小偷一脸沮丧回答道。

“想走？！可以。先买票再说！”售票员干脆地说。

小菜和大鸟对望一眼，异口同声道：“强！”

……

20.2 迭代器模式

“小菜，今天你真见到强人了吧。”大鸟在下车后，对小菜说道。

“这个售票员，实在够强，”小菜学着模仿道，“想走？！可以。先买票再说！”

“这售票员其实在做一件重要的事，就是把车厢里的所有人都遍历了一遍，不放过一个不买票的乘客。这也是一个设计模式的体现。”

“大鸟，你也够强，什么都可以往设计模式上套，这也是模式？”

“当然是模式。这个模式就叫做迭代器模式。”

> **迭代器模式（Iterator）**，提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。[DP]

“你想呀，售票员才不管你上来的是人还是物（行李），不管是中国人还是外国人，不管是不是内部员工，甚至哪怕是马上要抓走的小偷，只要是来乘车的乘客，就必须要买票。同样道理，当你需要访问一个聚集对象，而且不管这些对象是什么都需要遍历的时候，你就应该考虑用迭代器模式。另外，售票员从车头到车尾来售票，也可以从车尾向车头来售票，也就是说，你需要对聚集有多种方式遍历时，可以考虑用迭代器模式。由于不管乘客是什么，售票员的做法始终是相同的，都是从第一个开始，下一个是谁，是否结束，当前售到哪个人了，这些方法每天他都在做，也就是说，为遍历不同的聚集结构提供如开始、下一个、是否结束、当前哪一项等统一的接口。”

“听你这么一说，好像这个模式也不简单哦。”

“哈，本来这个模式还是有点意思的，不过现今来看迭代器模式实用价值远不如学习价值大了，MartinFlower甚至在自己的网站上提出撤销此模式。因为现在高级编程语言如C#、JAVA等本身已经把这个模式做在语言中了。”

“哦，是什么？”

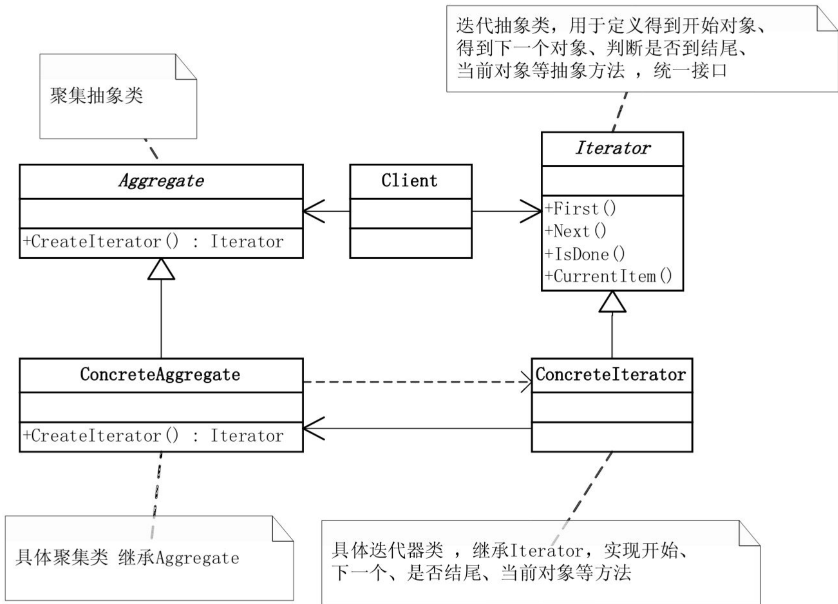
“哈，foreach in你熟悉吗？”

“啊，原来是它，没错没错，它就是不需要知道集合对象是什么，就可以遍历所有的对象的循环工具，非常好用。”

“另外还有像IEnumerable接口也是为迭代器模式而准备的。不管如何，学习一下GoF的迭代器模式的基本结构，还是很有学习价值的。研究历史是为了更好地迎接未来。”

20.3 迭代器实现

迭代器模式（Iterator）结构图



Iterator迭代器抽象类

```
abstract class Iterator
{

    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}
```

用于定义得到开始对象、得到下一个对象、判断是否到结尾、当前对象等抽象方法，统一接口

Aggregate聚集抽象类

```
abstract class Aggregate
{
    public abstract Iterator CreateIterator();
}
```

创建迭代器

ConcreteIterator具体迭代器类，继承Iterator

```
class ConcreteIterator : Iterator
{
    private ConcreteAggregate aggregate;
    private int current = 0;

    public ConcreteIterator(ConcreteAggregate aggregate)
    {
        this.aggregate = aggregate;
    }

    public override object First()
    {
        return aggregate[0];
    }

    public override object Next()
    {
        object ret = null;
        current++;
        if (current < aggregate.Count)
        {
            ret = aggregate[current];
        }
        return ret;
    }

    public override bool IsDone()
    {
        return current >= aggregate.Count ? true : false;
    }

    public override object CurrentItem()
    {
        return aggregate[current];
    }
}
```

定义了一个具体聚集对象

初始化时将具体的聚集对象传入

得到聚集的第一个对象

得到聚集的下一个对象

判断当前是否遍历到结尾，到结尾返回 true

返回当前的聚集对象

ConcreteAggregate具体聚集类 继承Aggregate

```
class ConcreteAggregate : Aggregate
{
    private IList<object> items = new List<object>();
    public override Iterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }

    public int Count
    {
        get { return items.Count; }
    }

    public object this[int index]
    {
        get { return items[index]; }
        set { items.Insert(index, value); }
    }
}
```

声明一个 IList 泛型变量，用于存放聚合对象，用 ArrayList 同样可以实现

返回聚集总个数

声明一个索引器

客户端代码

```
static void Main(string[] args)
{
    ConcreteAggregate a = new ConcreteAggregate();

    a[0] = "大鸟";
    a[1] = "小菜";
    a[2] = "行李";
    a[3] = "老外";
    a[4] = "公交内部员工";
    a[5] = "小偷";

    Iterator i = new ConcreteIterator(a);
    object item = i.First();
    while (!i.IsDone())
    {
        Console.WriteLine("{0} 请买车票!", i.CurrentItem());
        i.Next();
    }

    Console.Read();
}
```

公交车，即聚集对象

新上来的乘客，即对象数组

售票员出场，先看好了上车的是哪些人，即声明了迭代器对象

从第一个乘客开始

对面前的乘客告知请买票

下一乘客

运行结果

大鸟　请买车票
小菜　请买车票
行李　请买车票
老外　请买车票
公交内部员工　请买车票
小偷　请买车票

“看到没有，这就是我们的优秀售票员售票——迭代器的整个运作模式。”

“大鸟，你说为什么要用具体的迭代器ConcreteIterator来实现抽象的Iterator呢？我感觉这里不需要抽象呀，直接访问ConcreteIterator不是更好吗？”

“哈，那是因为刚才有一个迭代器的好处你没注意，当你需要对聚集有多种方式遍历时，可以考虑用迭代器模式，事实上，售票员一定要从车头到车尾这样售票吗？”

“你意思是，他还可以从后向前遍历？”

“当然是可以，你不妨再写一个实现从后往前的具体迭代器类看看。”

“好的。”

```
class ConcreteIteratorDesc : Iterator
{
    private ConcreteAggregate aggregate;
    private int current = 0;

    public ConcreteIteratorDesc(ConcreteAggregate aggregate)
    {
        this.aggregate = aggregate;
        current = aggregate.Count - 1;
    }
    public override object First()
    {
        return aggregate[aggregate.Count - 1];
    }
    public override object Next()
    {
        object ret = null;
        current--;
        if (current >= 0)
        {
            ret = aggregate[current];
        }
        return ret;
    }

    public override object CurrentItem()
    {
        return aggregate[current];
    }

    public override bool IsDone()
    {
        return current < 0 ? true : false;
    }
}
```

定义了一个具体聚集对象

初始化时将具体的聚集对象传入

得到聚集的第一个对象

得到聚集的下一个对象

判断当前是否遍历到结尾，到结尾返回 true

返回当前的聚集对象

“写得不错，这时你客户端只需要更改一个地方就可以实现反向遍历了”

```
//Iterator i = new ConcreteIterator(a);
Iterator i = new ConcreteIteratorDesc(a);
```

“是呀，其实售票员完全可以更多的方式来遍历乘客，比如从最高的到最矮、从最小到最老、从最靓丽酷毙到最猥琐龌龊。”小菜已经开始头脑风暴。

“神经病，你当是你呀。”大鸟笑骂。

20.4 .NET的迭代器实现

“刚才我们也说过，实际使用当中是不需要这么麻烦的，因为.NET框架已经为你准备好了相关接口，你只需去实现就好。”

IEumerator支持对非泛型集合的简单迭代接口。

```
public interface IEumerator
{
    object Current          // 获取集合中的当前元素
    {
        get;
    }
    bool MoveNext();        // 将枚举数推进到集合的下一个元素。方法返回值 true 表示迭代器成功前进到集合中的下一个元素，返回值 false 表示已经位于集合的末尾
    void Reset();           // 恢复初始化指向的位置，该位置位于集合中第一个元素之前
}
```

IEnumerable公开枚举数，该枚举数支持在非泛型集合上进行简单迭代。

```
public interface IEnumerable
{
    IEumerator GetEnumerator();   // 返回一个循环访问集合的枚举数
}
```

“你会发现，这两个接口，特别是IEumerator要比我们刚才写的抽象类Iterator要简洁，但可实现的功能却一点不少，这其实也是对GoF的设计改良的结果。”

“其实具体类实现这两个接口的代码也差别不大，是吗？”

“是的，区别不大，另外这两个接口还有相应的泛型接口，去查MSDN帮助就可以了。”

“有了这个基础，你再来看你最熟悉的foreach in就很简单了”。

```
static void Main(string[] args)
{
    IList<string> a=new List<string>();    // 也可以是 ArrayList 集合
    a.Add("大鸟");
    a.Add("小菜");
    a.Add("行李");
    a.Add("老外");
    a.Add("公交内部员工");
    a.Add("小偷");

    foreach (string item in a)
    {
        Console.WriteLine("{0} 请买车票!", item);
    }
    Console.Read();
}
```

“这里用到了foreach　in而在编译器里做了些什么呢？其实它做的是下面的工作。”

```
IEnumerator<string> e = a.GetEnumerator();

while(e.MoveNext())
{
    Console.WriteLine("{0} 请买车票!",
e.Current);
}
```

“原来foreach in就是实现这两个接口来实际循环遍历呀。”

“是的，尽管我们不需要显式的引用迭代器，但系统本身还是通过迭代器来实现遍历的。总地来说，迭代器（**Iterator**）模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。迭代器模式在访问数组、集合、列表等数据时，尤其是数据库数据操作时，是非常普遍的应用，但由于它太普遍了，所以各种高级语言都对它进行了封装，所以反而给人感觉此模式本身不太常用了。”

20.5 迭代高手

“哈哈，看来那个售票员是最了不起的迭代高手，每次有乘客上车他都数数，统计人数，然后再对整车的乘客进行迭代遍历，不放过任何漏网之鱼，啊，应该是逃票之人。”

“隔行如隔山，任何行业都有技巧和经验，需要多思考、多琢磨，才能做到最好的。”

“嗯，编程又何尝不是这样，我相信代码没有最好，只有更好，我要继续努力。”

第21章　有些类也需计划生育——单例模式

21.1　类也需要计划生育

时间：5月29日20点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“大鸟，今天我在公司写一个MDI窗体程序，当中有一个是‘工具箱’的窗体，问题就是，我希望工具箱要么不出现，出现也只出现一个，可实际上却是我每点击菜单，实例化‘工具箱’，它就会出来一个，这样点击多次就会出现很多个‘工具箱’，你说怎么办？”

“哈，显然，你的这个‘工具箱’类需要计划生育呀，你让它超生了，当然是不好的。”

“大鸟，你又在说笑了，不过计划生育的说法也算是贴切吧，现在我就是希望它要么不要有，有就只能一个，如何办？”

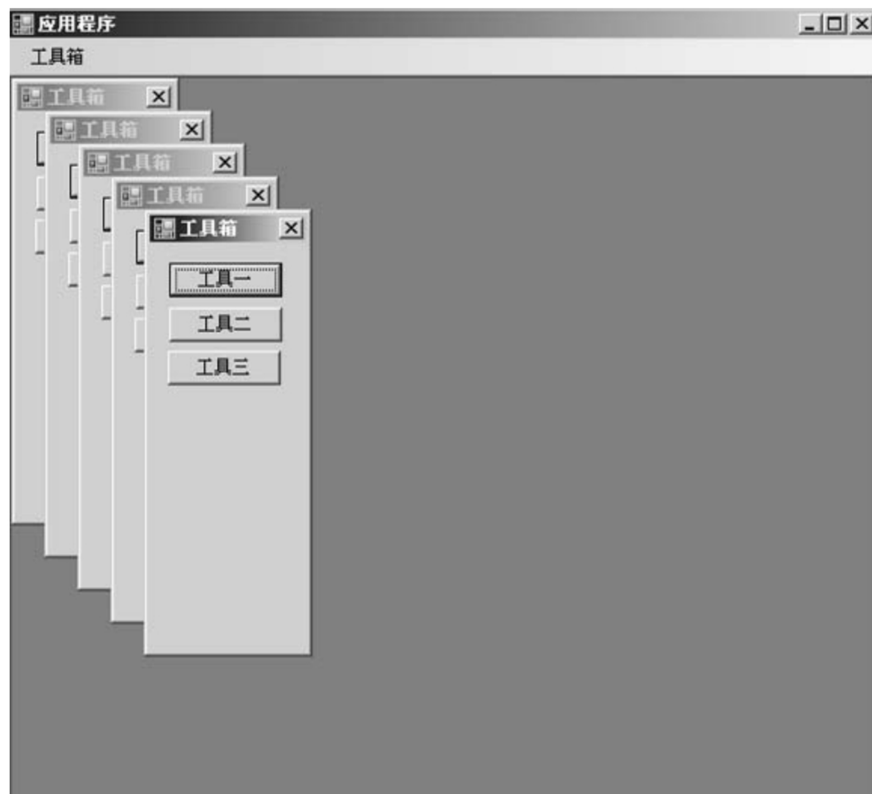
“其实这就是一个设计模式的应用，你先说说你是怎么写的？”

“代码是这样的，首先我建立了一个windows应用程序，默认的窗体为Form1.cs，我设置了它的IsMdiContainer属性为true，表示它是多文档界面MDI子窗体的容器。然后建立了一个菜单，菜单第一项ToolStripMenuItemToolbox就是启动‘工具箱’的按钮。我另建立一个窗体叫FormToolbox.cs，也就是‘工具箱’窗体，里面有一些相关工具按钮。”

```
private void Form1_Load(object sender, EventArgs e)
{
    this.IsMdiContainer = true;        // 也可以直接设置 Form1 窗体的 IsMdiContainer 属性为 true
}

private void ToolStripMenuItemToolbox_Click(object sender, EventArgs e)
{
    FormToolbox ftb = new FormToolbox ();   // “工具箱”启动的代码，实例化 FormToolbox，并设置其 Mdi 父窗体为 Form1
    ftb.MdiParent = this;
    ftb.Show();
}
```

代码执行后的样子如下



“我每点击一次‘工具箱’的菜单项，就产生一个新的‘工具箱’窗体，但实际上，我只希望它出现一次，或者干脆不出现。”

21.2 判断对象是否是null

“这个其实不难办呀，你判断一下，这个FormToolbox有没有实例化过不就行了。”

“什么叫FormToolbox有没有实例化过？我是在按了菜单的按钮时，才去FormToolbox ftb = new FormToolbox ( ) ;那当然是新实例化了。”

“问题就在于此呀，为什么要在点击按钮时才声明FormToolbox对象呢，你完全可以把声明的工作放到类的全局变量中完成。这样你就可以去判断这个变量是否被实例化过了。”

“哦，明白，原来如此。我改一改。”

```
private FormToolbox ftb;                    类变量声明

private void ToolStripMenuItemToolbox_Click(object sender, EventArgs e)
{
    if (ftb==null)                          判断是否实例化过，如果为
    {                                       null 说明没有实例化过
        ftb = new FormToolbox ();
        ftb.MdiParent = this;
        ftb.Show();
    }
}
```

“原来如此，就这么简单，好了，这个功能就算是完成了。”

“小菜，你也太知足了吧，这就算是好了？如果做任何事情不求完美，只求简单达成目标，那你又如何能有提高。”

“这样的一个小程序还可以再完善什么呢？”

“打个比方，你现在不但要在菜单里启动‘工具箱’，还需要在‘工具栏’上有一个按钮来启动‘工具箱’，你如何做？”

“这个不难，首先加一个工具栏控件toolStrip1，然后再在按钮toolStripButton1的click事件代码里写同样的代码就可以了。”



增加一个工具栏按钮的事件处理

```
private void toolStripButton1_Click(object sender,
EventArgs e)
{
    if(ftb == null)
```

```
        {
            ftb = new FormToolbox();
            ftb.MdiParent = this;
            ftb.Show();
        }
    }
```

“小菜，我还正想提醒你，复制粘贴是最容易的编程，但也是最没有价值的编程。你现在将两个地方的代码复制在一起，这就是重复。这要是需求变化或有Bug时就需要改多个地方。”

“哈，你说得也是，最好是提炼出一个方法来让他们调用。不过这里应该不会有什么变化的。”

“你就那么肯定不会有问题？”

“不会。”

“你把程序运行后，启动‘工具箱’，然后把‘工具箱’窗体关闭，你再点启动按钮看看。”

……

“啊，‘工具箱’怎么就不出现了？”

“哼哼，原因是当你关闭‘工具箱’时，它的实例并没有变为null而只是Disposed。你的判断只是是否等于null，当然就不会再实例化了。”

“哦，原来是需要再增加IsDisposed属性的判断。”

```
private void toolStripButton1_Click(object sender, EventArgs e)
{
    if (ftb == null || ftb.IsDisposed)
    {
        ftb = new FormToolbox();
        ftb.MdiParent = this;
        ftb.Show();
    }
}
```

“如果我现在有五个地方需要实例化出‘工具箱’窗体，这个小bug就需要改动五个地方，你说复制粘贴害不害人？”

“是的，那我将它提炼到一个方法里去就行了。”

```
private FormToolbox ftb;

private void ToolStripMenuItemToolbox_Click(object sender, EventArgs e)
{
    openToolbox();
}

private void toolStripButton1_Click(object sender, EventArgs e)
{
    openToolbox();
}

private void openToolbox()
{
    if (ftb == null || ftb.IsDisposed)
    {
        ftb = new FormToolbox();
        ftb.MdiParent = this;
        ftb.Show();
    }
}
```

提炼出“打开工具箱”的方法

“现在好了吧，应该没什么大问题了。”

21.3 生还是不生是自己的责任

“有，当然还有。夫妻已经有了一个小孩子，下面是否生第二胎，这是谁来负责呀？”

“当然是他们自己负责，要是超生了，违反了国家的政策，那也是他们自己的原因。”

“说得好，你再想想看这种场景：领导问下属，报告交了没有，下属可能说‘早交了’于是领导满意地点点头，下属也可能说‘还剩下一点内容没写，很快上交’，领导皱起眉头说‘要抓紧’。此时这份报告交还是没交，由谁来判断？”

“当然是下属自己的判断，因为下属最清楚报告交了没有，领导只需要问问就行了。”

“好了，同样的，现在‘工具箱’FormToolbox是否实例化都是在MDI主窗体Form1的代码里判断，你不觉得这不合逻辑吗？”

“你的意思是说，Form1里应该只是通知启动‘工具箱’，至于‘工具箱’窗体是否实例化过，应该由‘工具箱’自己来判断？”

“哈，当然，实例化与否的过程其实就和报告交了与否的过程一样，应该由自己来判断，这是它自己的责任，而不是别人的责任。别人应该只是使用它就可以了。”

“哦，我想想看，实例化其实就是new的过程，但问题就是我怎么让人家不用new呢？”

“是的，如果你不对构造方法做改动的话，是不可能阻止他人不去用new的。所以我们完全可以直接就把这个类的构造方法改成私有（private），你应该知道，所有类都有构造方法，不编码则系统默认生成空的构造方法，若有显示定义的构造方法，默认的构造方法就会失效。于是只要你将‘工具箱’类的构造方法写成是private的，那么外部程序就不能用new来实例化它了。”

“哈，私有的方法外界不能访问，这是对的，但是这样一来，这个类如何能有实例呢？”

“哈，我们的目的是什么？”

“让这个类只能实例化一次。没有new，我现在连一次也不能实例化了。”

“错，只能说，对于外部代码，不能用new来实例化它，但是我们完

全可以再写一个public方法，叫做GetInstance（），这个方法的目的就是返回一个类实例，而此方法中，去做是否有实例化的判断。如果没有实例化过，由调用private的构造方法new出这个实例，之所以它可以调用是因为它们在同一个类中，private方法可以被调用的。”

“不是很懂，你把代码写出来吧。”

“好的，你看……”

```
public partial class FormToolbox : Form
{
    private static FormToolbox ftb = null;

    private FormToolbox()
    {
        InitializeComponent();
    }
    public static FormToolbox GetInstance()
    {
        if (ftb == null || ftb.IsDisposed)
        {
            ftb = new FormToolbox();
            ftb.MdiParent=Form1.ActiveForm;
        }
        return ftb;
    }
}
```

声明一个静态的类变量

构造方法私有，外部代码不能直接 new 来实例化它

得到类实例的方法，返回值就是本类对象，注意也是静态的

当内部的 ftb 是 null 或者被 Dispose 过，则 new 它。并且设计其 MdiParent 为 Form1，此时将实例化的对象存在静态的变量 ftb 中，以后就可以不用实例而得到它了

“其实也就是把你之前写的代码搬到了‘工具箱’FormToolbox中，由于构造方法私有，就只能从内部去调用。然后当访问静态的公有方法GetInstance（）时，它会先去查看内存中有没有这个类的实例，若有就直接返回，也就是不会超生了。”

“哦，我知道了。就拿计划生育的例子来说，刚解放时，国家需要人，人多力量大嘛，于是老百姓生！生！生！于是人口爆炸了。后来实行了计划生育，规定了一对夫妇最多只能生育一胎，并把判断的责任交给了夫妇，于是刚结婚时，想要孩子就生一个，而生好一个后，无论谁来要求，都不生了，因为有一个孩子，不可以再生了，否则无论对家庭还是国家都将是沉重的负担。”

“有点偏激，但也可以这么理解吧。客户端的代码如何写呢？”

“好说，好说！这就不难了。”

```
private void ToolStripMenuItemToolbox_Click(object se
EventArgs e)
```

```
{
    FormToolbox.GetInstance().Show();
}

private void toolStripButton1_Click(object sender,
EventArgs e)
{
    FormToolbox.GetInstance().Show();
}
```

“这样一来，客户端不再考虑是否需要去实例化的问题，而把责任都给了应该负责的类去处理。其实这就是一个很基本的设计模式：单例模式。”

21.4 单例模式

> **单例模式（Singleton）**，保证一个类仅有一个实例，并提供一个访问它的全局访问点。[DP]

“通常我们可以让一个全局变量使得一个对象被访问，但它不能防止你实例化多个对象。一个最好的办法就是，让类自身负责保存它的唯一实例。这个类可以保证没有其他实例可以被创建，并且它可以提供一个访问该实例的方法。**[DP]**”

单例模式（**Singleton**）结构图

Singleton
-instance : Singleton
-Singleton () +GetInstance ()

Singleton 类，定义一个GetInstance 操作，允许客户访问它的唯一实例。GetInstance 是一个静态方法，主要负责创建自己的唯一实例

Singleton类，定义一个GetInstance操作，允许客户访问它的唯一实例。GetInstance是一个静态方法，主要负责创建自己的唯一实例。

```
class Singleton
{
    private static Singleton instance;

    private Singleton()        // 构造方法让其 private，这就堵死了外界利用 new 创建此类实例的可能
    {
    }

    public static Singleton GetInstance()        // 此方法是获得本类实例的唯一全局访问点
    {
        if (instance == null)        // 若实例不存在，则 new 一个新实例，否则返回已有的实例
        {
            instance = new Singleton();
        }

        return instance;
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    Singleton s1 = Singleton.GetInstance();
    Singleton s2 = Singleton.GetInstance();

    if (s1 == s2)
    {
        Console.WriteLine("两个对象是相同的实例。");
    }

    Console.Read();
}
```

比较两次实例化后对象的结果是实例相同

“单例模式除了可以保证唯一的实例外，还有什么好处呢？”

“好处还有呀，比如单例模式因为**Singleton**类封装它的唯一实例，这样它可以严格地控制客户怎样访问它以及何时访问它。简单地说就是对唯一实例的受控访问。”

“我怎么感觉单例有点像一个实用类的静态方法，比如.Net框架里的Math类，有很多数学计算方法，这两者有什么区别呢？”

“你说得没错，它们之间的确很类似，实用类通常也会采用私有化的构造方法来避免其有实例。但它们还是有很多不同的，比如实用类不保存状态，仅提供一些静态方法或静态属性让你使用，而单例类是有状态的。实用类不能用于继承多态，而单例虽然实例唯一，却是可以有子类来继承。实用类只不过是一些方法属性的集合，而单例却是有着唯一的对象实例。在运用中还得仔细分析再作决定用哪一种方式。”

“哦，我明白了。”

21.5　多线程时的单例

“另外，你还需要注意一些细节，比如说，多线程的程序中，多个线程同时，注意是同时访问Singleton类，调用GetInstance（）方法，会有可能造成创建多个实例的。”

“啊，是呀，这应该怎么办呢？”

“可以给进程一把锁来处理。这里需要解释一下lock语句的涵义，**lock**是确保当一个线程位于代码的临界区时，另一个线程不进入临界区。如果其他线程试图进入锁定的代码，则它将一直等待（即被阻止），直到该对象被释放。**[MSDN]**”

```
class Singleton
{
    private static Singleton instance;
    private static readonly object syncRoot= new object();   // 程序运行时创建一个静态只读的进程辅助对象
    private Singleton()
    {
    }

    public static Singleton GetInstance()
    {
        lock (syncRoot)   // 在同一个时刻加了锁的那部分程序只有一个线程可以进入
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
        }
        return instance;
    }
}
```

“这段代码使得对象实例由最先进入的那个线程创建，以后的线程在进入时不会再去创建对象实例了。由于有了lock，就保证了多线程环境下的同时访问也不会造成多个实例的生成。”

“为什么不直接lock（instance），而是再创建一个syncRoot来lock呢？”

“小菜呀，加锁时，instance实例有没有被创建过实例都还不知道，怎么对它加锁呢？”

“我知道了，原来是这样。但这样就得每次调用GetInstance方法时都需要lock，好像不太好吧。”

“说得非常好，的确是这样，这种做法是会影响性能的，所以对这个类再做改良。”

21.6 双重锁定

```
class Singleton
{
    private static Singleton instance;
    private static readonly object syncRoot= new object();
    private Singleton()
    {
    }

    public static Singleton GetInstance()
    {
        if (instance == null)
        {
            lock (syncRoot)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

先判断实例是否存在，不存在再加锁处理

“现在这样，我们不用让线程每次都加锁，而只是在实例未被创建的时候再加锁处理。同时也能保证多线程的安全。这种做法被称为Double-Check Locking（双重锁定）。”

“我有问题，我在外面已经判断了instance实例是否存在，为什么在lock里面还需要做一次instance实例是否存在的判断呢？”小菜问道。

```
............
public static Singleton GetInstance()
{
```

```
        if (instance == null)
        {
            lock (syncRoot)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
    ............
```

“那是因为你没有仔细分析。对于instance存在的情况，就直接返回，这没有问题。当instance为null并且同时有两个线程调用GetInstance（）方法时，它们将都可以通过第一重instance==null的判断。然后由于lock机制，这两个线程则只有一个进入，另一个在外排队等候，必须要其中的一个进入并出来后，另一个才能进入。而此时如果没有了第二重的instance是否为null的判断，则第一个线程创建了实例，而第二个线程还是可以继续再创建新的实例，这就没有达到单例的目的。你明白了吗？”

“哦，明白了，原来有这么麻烦呀。”

21.7 静态初始化

“其实在实际应用当中，C#与公共语言运行库也提供了一种‘静态初始化’方法，这种方法不需要开发人员显式地编写线程安全代码，即可解决多线程环境下它是不安全的问题。**[MSDN]**”

“哦，还有更好的办法？”

“谈不上更好，只不过实现更简单。我们来看代码。”

```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton GetInstance()
    {
        return instance;
    }
}
```

阻止发生派生，而派生可能会增加实例

在第一次引用类的任何成员时创建实例。公共语言运行库负责处理变量初始化

“这样的实现与前面的示例类似，也是解决了单例模式试图解决的两个基本问题：全局访问和实例化控制，公共静态属性为访问实例提供了一个全局访问点。不同之处在于它依赖公共语言运行库来初始化变量。由于构造方法是私有的，因此不能在类本身以外实例化Singleton类；因此，变量引用的是可以在系统中存在的唯一的实例。不过要注意，instance变量标记为readonly，这意味着只能在静态初始化期间或在类构造函数中分配变量[MSDN]。由于这种静态初始化的方式是在自己被加载时就将自己实例化，所以被形象地称之为饿汉式单例类，原先的单例模式处理方式是要在第一次被引用时，才会将自己实例化，所以就被称为懒汉式单例类。**[J&DP]**”

“懒汉饿汉，哈，很形象的比喻。它们主要有什么区别呢？”

“由于饿汉式，即静态初始化的方式，它是类一加载就实例化的对象，所以要提前占用系统资源。然而懒汉式，又会面临着多线程访问的安全性问题，需要做双重锁定这样的处理才可以保证安全。所以到底使用哪一种方式，取决于实际的需求。从C#语言角度来讲，饿汉式的单例类已经足够满足我们的需求了。

“没想到小小的单例模式也有这么多需要考虑的问题。”小菜感叹道。

“刚接触时，都会觉得比较复杂，但其实用多了，也就这样了。就像70年代末时，老百姓都不太能接受一对夫妇只生一个小孩的计划生育政策，但是随着这近三十年的社会发展，教育和生活成本的大大提高，生

两个都养不起，别说七个八个。甚至还干脆出现了‘丁客’一族，不生了，有钱自己用。这在以前能想象吗？”大鸟类比说。

“哈，这都哪跟哪呀，计划生育和单例模式根本就是两回事，大鸟又在胡说八道。好了，我再去研究一下单例的程序，bye-bye。”

第22章　手机软件何时统一——桥接模式

22.1　凭什么你的游戏我不能玩

时间：5月31日20点　　地点：大鸟房间　　人物：小菜、大鸟

“大鸟，捧着个手机，玩什么呢？”小菜冲进了大鸟的房门。

“哈，玩小游戏呢，新买的手机，竟然可以玩小时候的游戏‘魂斗罗’。很久没碰这东西了，感觉很爽哦。”

“哦，是吗，连这游戏都有呀，给我看看。”

“等等，等我死了再说。”大鸟玩得正开心。

“等你死了？”小菜笑道，“你什么时候会‘死’呀？”

“那还有段时间了，至少半小时吧。”

“半小时才死呀，哦，那我半小时后来给你收尸。”小菜故意提高嗓门。

“你小子，找死呀。给你给你！”大鸟笑着把手机递给了小菜，“游戏和红白机上的一模一样，很让人怀旧呀。嗨，我跟你们这种80后小子说红白机，不就等于对牛弹琴吗！”

“怎么没玩过，我可也是任天堂红白机高手哦。大鸟别把我想得好像和你不是一代人一样，我们的童年应该差不多的。”

“现在时代变化太快了，差五岁，差不多就是差一代人，你是80后，我是70后，我们的童年差距当然很大。”

“哪有这么严重，‘魂斗罗’也是我很喜欢的游戏。对了，这游戏可以装到我的手机里吗？”

“你的手机是M品牌的吧，我的是N品牌的，按道理我这里的游戏你是不能玩的。”

“是吗，这真是太扫兴了。你说这手机为什么不能统一一下软件呢？”

“其实手机真正的发展也就近十年，此期间各大手机厂商都发展自己的软件部门开发手机软件，哪怕是同一品牌的手机，不同型号的也完全有可能软件不兼容。”

“是的是的，”小菜点头道，“我以前用过的N品牌的两款手机，功能都是固化在手机里的，不同手机差别太大了，最早那个手机的拼音输入法实在是傻得要死，要发个短信得输入半天，和现在的输入法比真可说是天壤之别。现在好一点，同品牌的新手机，型号不同，软件还算是基本兼容，可惜不同品牌，软件基本还是不能整合在一起。”

“但你有没有想过，在计算机领域里，就完全不一样了。比如由于有了Windows操作系统，使得所有的PC厂商不用关注软件，而软件制造商也不用过多关注硬件，这对计算机的整体发展是非常有利的。而有个别品牌的电脑公司自己开发操作系统和应用软件，尽管充满了创意，但却因为不能与其他软件整合，而使得发展缓慢，连盗版都不愿意光顾它。”

“哈，手机为什么不可以学计算机呢？由专业公司开发操作系统和应用软件，手机商只要好好把手机硬件做好就行了。”

“统一谈何容易，谁做的才算是标准呢？而谁又不希望自己的硬件和软件成为标准，然后一统天下。这里有很多商业竞争的问题，不是我们想的这么简单。不过目前很多智能手机都在朝这个方向发展。或许过几年，我们手里的机器就可以实现软件完全兼容了。”

“我想那时应该就不叫做手机了，而是掌上电脑才更合适。”

22.2　紧耦合的程序演化

“说得有道理，另外你有没有想过，这里其实蕴含两种完全不同的思维方式？”

“你是说手机硬件软件和PC硬件软件？”

“对的，如果我现在有一个N品牌的手机，它有一个小游戏，我要玩游戏，程序应该如何写？”

“这还不简单。先写一个此品牌的游戏类，再用客户端调用即可。”

游戏类

```
//N品牌的手机中的游戏
class HandsetNGame
{
    public void Run()
    {
        Console.WriteLine("运行N品牌手机游戏");
    }
}
```

客户端代码

```
HandsetNGame game=new HandsetNGame();
game.Run();
```

“很好，现在又有一个M品牌的手机，也有小游戏，客户端也可以调用，如何做？”

“嗯，我想想，两个品牌，都有游戏，我觉得从面向对象的思想来说，应该有一个父类‘手机品牌游戏’，然后让N和M品牌的手机游戏都继承于它，这样可以实现同样的运行方法”。

“小菜不错，抽象的感觉来了。”

手机游戏类

```
class HandsetGame
{
    public virtual void Run()
    {
    }
}
```

M品牌手机游戏和N品牌手机游戏

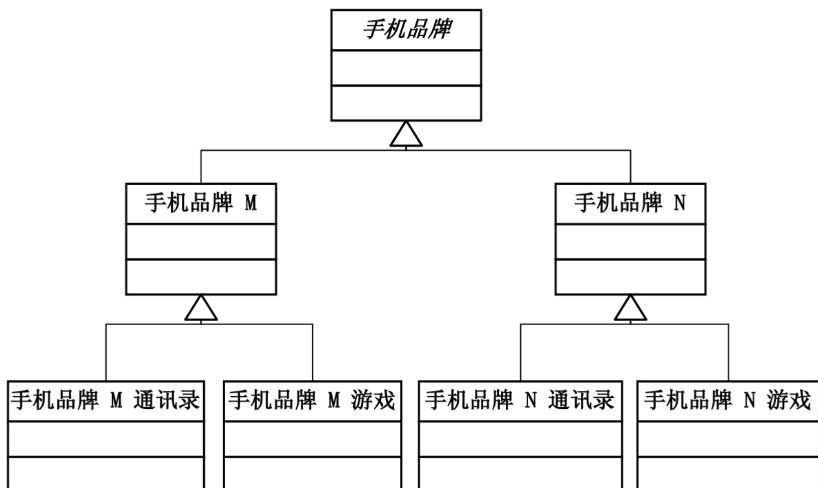
```
class HandsetMGame : HandsetGame
{
    public override void Run()
    {
        Console.WriteLine("运行M品牌手机游戏");
    }
}

class HandsetNGame : HandsetGame
{
    public override void Run()
    {
        Console.WriteLine("运行N品牌手机游戏");
    }
}
```

“然后，由于手机都需要通讯录功能，于是N品牌和M品牌都增加了通讯录的增删改查功能。你如何处理？”

“啊，这就有点麻烦了，那就意味着，父类应该是‘手机品牌’，下有‘手机品牌M’和‘手机品牌N’，每个子类下各有‘通讯录’和‘游戏’子类。”

代码结构图



手机类

```
//手机品牌
class HandsetBrand
{
    public virtual void Run()
    {
    }
}
```

手机品牌N和手机品牌M类

```
//手机品牌M
class HandsetBrandM : HandsetBrand
{
}

//手机品牌N
class HandsetBrandN : HandsetBrand
{
}
```

下属的各自通讯录类和游戏类

```
//手机品牌M的游戏
class HandsetBrandMGame : HandsetBrandM
{
    public override void Run()
    {
        Console.WriteLine("运行M品牌手机游戏");
    }
}
//手机品牌N的游戏
class HandsetBrandNGame : HandsetBrandN
{
    public override void Run()
    {
        Console.WriteLine("运行N品牌手机游戏");
    }
}
//手机品牌M的通讯录
class HandsetBrandMAddressList : HandsetBrandM
{
    public override void Run()
    {
        Console.WriteLine("运行M品牌手机通讯录");
    }
}
//手机品牌N的通讯录
class HandsetBrandNAddressList : HandsetBrandN
{
    public override void Run()
    {
        Console.WriteLine("运行N品牌手机通讯录");
    }
}
```

客户端调用代码

```
static void Main(string[] args)
{
    HandsetBrand ab;

    ab = new HandsetBrandMAddressList();
    ab.Run();
```

```
        ab = new HandsetBrandMGame();
        ab.Run();

        ab = new HandsetBrandNAddressList();
        ab.Run();

        ab = new HandsetBrandNGame();
        ab.Run();

        Console.Read();
    }
```

“哈，这个结构应该还是可以的，现在我问你，如果我现在需要每个品牌都增加一个MP3音乐播放功能，你如何做？”

“这个？那就在每个品牌的下面都增加一个子类。”

“你觉得这两个子类差别大不大？”大鸟追问道。

“应该是不大的，不过没办法呀，因为品牌不同，增加功能就必须要这样的。”小菜无奈地说。

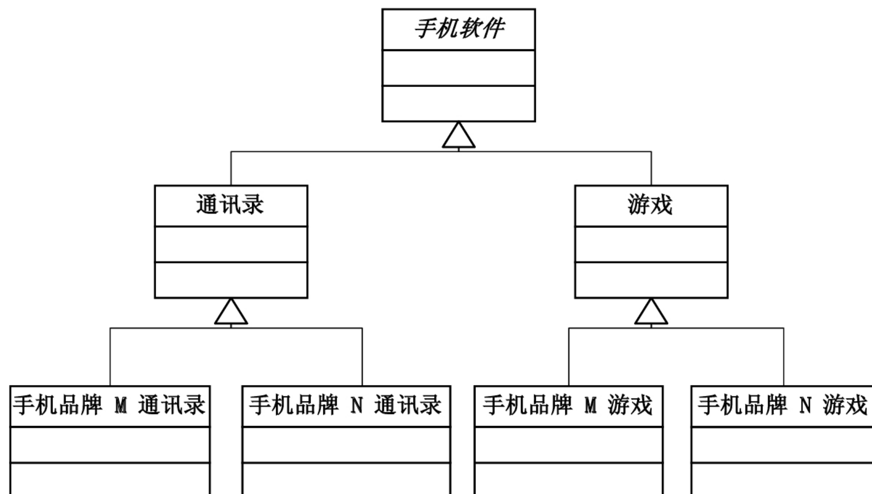
“好，那我现在又来了一家新的手机品牌‘S’，它也有游戏、通讯录、MP3音乐播放功能，你如何处理？”

“啊，那就得再增加‘手机品牌S’类和三个下属功能子类。这好像有点麻烦了。”

“你也感觉麻烦啦？如果我还需要增加‘输入法’功能、‘拍照’功能，再增加‘L品牌’、‘X品牌’你的类如何写？”

“啊哦，”小菜学了一声唐老鸭的叫声，感慨道，“我要疯了。要不这样，我换一种方式。”

过了几分钟，小菜画出了另一种结构图。



“你觉得这样子问题就可以解决吗？”

“啊，”小菜摇了摇头，“不行，要是增加手机功能或是增加品牌都会产生很大的影响。”

“你知道问题出在哪里吗？”

“我不知道呀，”小菜很疑惑，“我感觉我一直在用面向对象的理论设计的，先有一个品牌，然后多个品牌就抽象出一个品牌抽象类，对于每个功能，就都继承各自的品牌。或者，不从品牌，从手机软件的角度去分类，这有什么问题呢？”

“是呀，就像我刚开始学会用面向对象的继承时，感觉它既新颖又功能强大，所以只要可以用，就都用上继承。这就好比是‘有了新锤子，所有的东西看上去都成了钉子。[DPE]’但事实上，很多情况用继承会带来麻烦。比如，对象的继承关系是在编译时就定义好了，所以无法在运行时改变从父类继承的实现。子类的实现与它的父类有非常紧密的依赖关系，以至于父类实现中的任何变化必然会导致子类发生变化。当你需要复用子类时，如果继承下来的实现不适合解决新的问题，则父类必须重写或被其他更适合的类替换。这种依赖关系限制了灵活性并最终限制了复用性 **[DP]** 。”

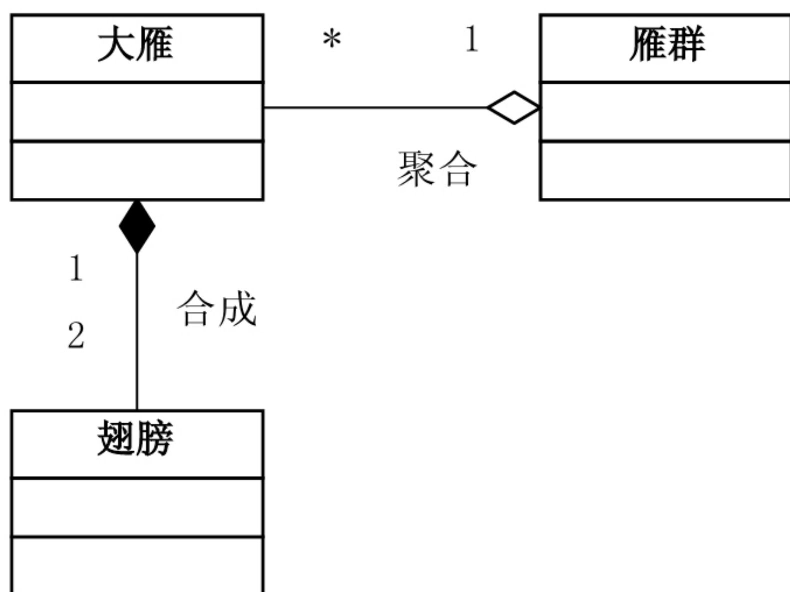
“是呀，我这样的继承结构，如果不断地增加新品牌或新功能，类会越来越多的。”

“在面向对象设计中，我们还有一个很重要的设计原则，那就是合成/聚合复用原则。即优先使用对象合成/聚合，而不是类继承[DP]。

22.3 合成/聚合复用原则

> 合成/聚合复用原则（**CARP**），尽量使用合成/聚合，尽量不要使用类继承。[J&DP]

合成（Composition，也有翻译成组合）和聚合（Aggregation）都是关联的特殊种类。聚合表示一种弱的'拥有'关系，体现的是**A**对象可以包含**B**对象，但**B**对象不是**A**对象的一部分；合成则是一种强的'拥有'关系，体现了严格的部分和整体的关系，部分和整体的生命周期一样 **[DPE]**。比方说，大雁有两个翅膀，翅膀与大雁是部分和整体的关系，并且它们的生命周期是相同的，于是大雁和翅膀就是合成关系。而大雁是群居动物，所以每只大雁都是属于一个雁群，一个雁群可以有多只大雁，所以大雁和雁群是聚合关系。"



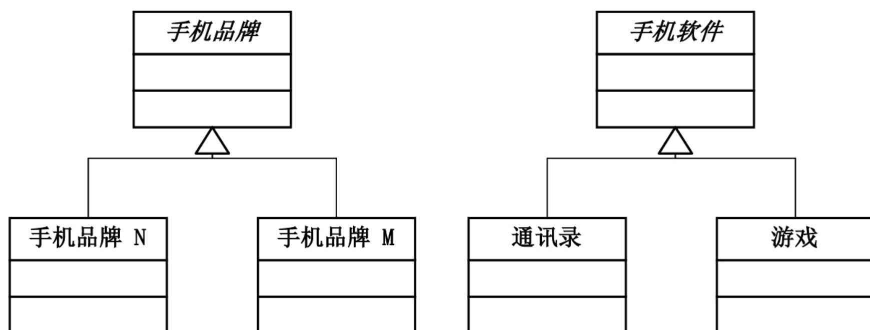
"合成/聚合复用原则的好处是，优先使用对象的合成/聚合将有助于你保持每个类被封装，并被集中在单个任务上。这样类和类继承层次会保持较小规模，并且不太可能增长为不可控制的庞然大物 **[DP]**。就刚才的例子，你需要学会用对象的职责，而不是结构来考虑问题。其实答案就在之前我们聊到的手机与PC电脑的差别上。"

“哦，我想想看，手机是不同的品牌公司，各自做自己的软件，就像我现在的设计一样，而PC却是硬件厂商做硬件，软件厂商做软件，组合起来才是可以用的机器。你是这个意思吗？”

“很好，我很喜欢你提到的‘组合’这个词，实际上，像‘游戏’、‘通讯录’、‘MP3音乐播放’这些功能都是软件，如果我们可以让其分离与手机的耦合，那么就可以大大减少面对新需求时改动过大的不合理情况。”

“好的好的，我想想怎么弄，你的意思其实就是应该有个‘手机品牌’抽象类和‘手机软件’抽象类，让不同的品牌和功能都分别继承于它们，这样要增加新的品牌或新的功能都不用影响其他类了。”

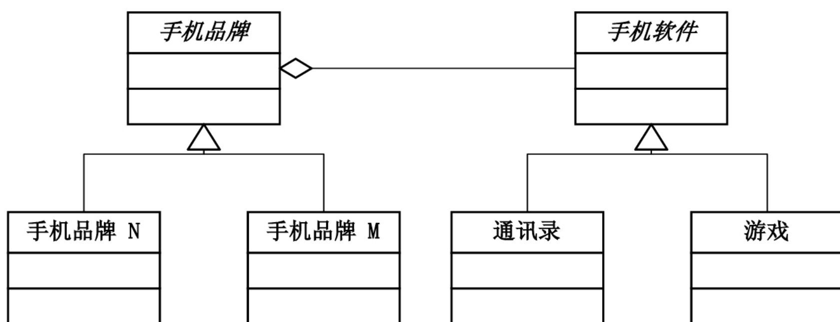
结构图



“还剩个问题，手机品牌和手机软件之间的关系呢？”大鸟问道。

“我觉得应该是手机品牌包含有手机软件，但软件并不是品牌的一部分，所以它们之间是聚合关系。”

结构图



“说得好。来试着写写看吧。”

22.4　松耦合的程序

小菜经过半小时，改动代码如下。

手机软件抽象类

```
//手机软件
abstract class HandsetSoft
{
    public abstract void Run();
}
```

游戏、通讯录等具体类

```
//手机游戏
class HandsetGame : HandsetSoft
{
    public override void Run()
    {
        Console.WriteLine("运行手机游戏");
    }
}

//手机通讯录
class HandsetAddressList : HandsetSoft
{
    public override void Run()
    {
        Console.WriteLine("运行手机通讯录");
    }
}
```

手机品牌类

```
//手机品牌
abstract class HandsetBrand
{
    protected HandsetSoft soft;

    //设置手机软件
    public void SetHandsetSoft(HandsetSoft soft)
    {
        this.soft = soft;
    }
    //运行
    public abstract void Run();
}
```

品牌需要关注软件，所以可在机器中安装软件（设置手机软件），以备运行

品牌N品牌M具体类

```
//手机品牌N
class HandsetBrandN : HandsetBrand
{
    public override void Run()
    {
        soft.Run();
    }
}

//手机品牌M
class HandsetBrandM : HandsetBrand
{
    public override void Run()
    {
        soft.Run();
    }
}
```

客户端调用代码

```
static void Main(string[] args)
{
    HandsetBrand ab;
    ab = new HandsetBrandN();

    ab.SetHandsetSoft(new HandsetGame());
    ab.Run();
```

```
        ab.SetHandsetSoft(new HandsetAddressList());
        ab.Run();

        ab = new HandsetBrandM();

        ab.SetHandsetSoft(new HandsetGame());
        ab.Run();

        ab.SetHandsetSoft(new HandsetAddressList());
        ab.Run();

        Console.Read();
    }
```

“感觉如何？是不是好很多。”

“是呀，现在如果要增加一个功能，比如MP3音乐播放功能，那么只要增加这个类就行了。不会影响其他任何类。类的个数增加也只是一个。”

```
    //手机MP3播放
    class HandsetMP3 : HandsetSoft
    {
        public override void Run()
        {
            Console.WriteLine("运行手机MP3播放");
        }
    }
```

“如果是要增加S品牌，只需要增加一个品牌子类就可以了。个数也是一个，不会影响其他类的 改动。”

```
    //手机品牌S
    class HandsetBrandS : HandsetBrand
    {
        public override void Run()
        {
```

```
            soft.Run ( ) ;
        }
    }
```

“这显然是也符合了我们之前的一个什么设计原则？”

“开放-封闭原则。这样的设计显然不会修改原来的代码，而只是扩展类就行了。但今天我的感受最深的是合成/聚合复用原则，也就是优先使用对象的合成或聚合，而不是类继承。聚合的魅力无限呀。相比，继承的确很容易造成不必要的麻烦。”

“盲目使用继承当然就会造成麻烦，而其本质原因主要是什么？”

“我想应该是，继承是一种强耦合的结构。父类变，子类就必须要变。”

“OK，所以我们在用继承时，一定要在是‘is-a’的关系时再考虑使用，而不是任何时候都去使用。”

“大鸟，今天这个例子是不是一个设计模式？”

“哈，当然，你看看刚才画的那幅图，两个抽象类之间有什么，像什么？”

“有一个聚合线，哈，像一座桥。”

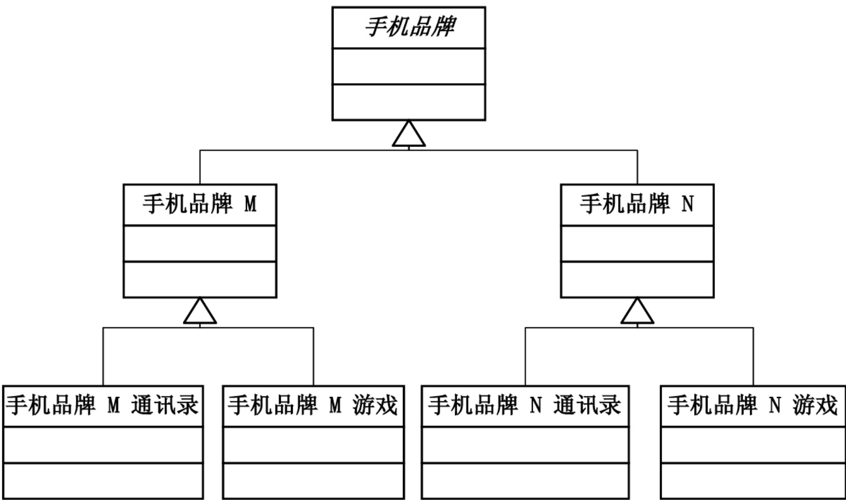
“好，说得好，这个设计模式就叫做‘桥接模式’。”

22.5 桥接模式

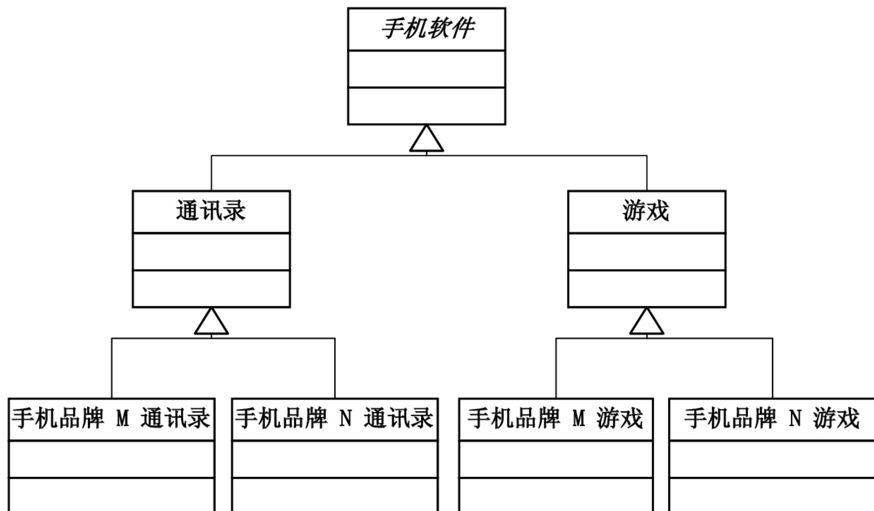
桥接模式（Bridge），将抽象部分与它的实现部分分离，使它们都可以独立地变化。[DP]

"这里需要理解一下，什么叫抽象与它的实现分离，这并不是说，让抽象类与其派生类分离，因为这没有任何意义。实现指的是抽象类和它的派生类用来实现自己的对象 **[DPE]** 。就刚才的例子而言，就是让'手机'既可以按照品牌来分类，也可以按照功能来分类。"

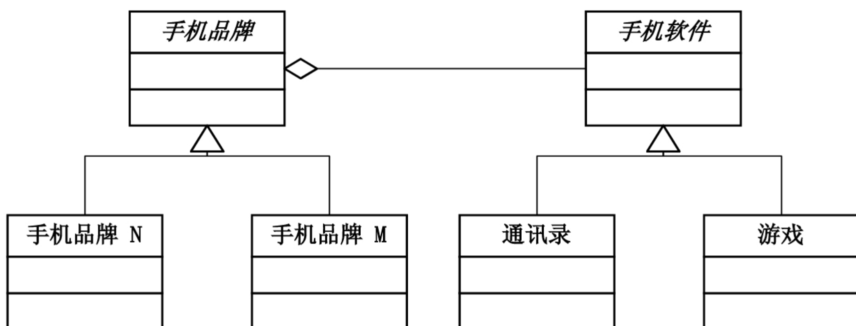
按品牌分类实现结构图



按软件分类实现结构图

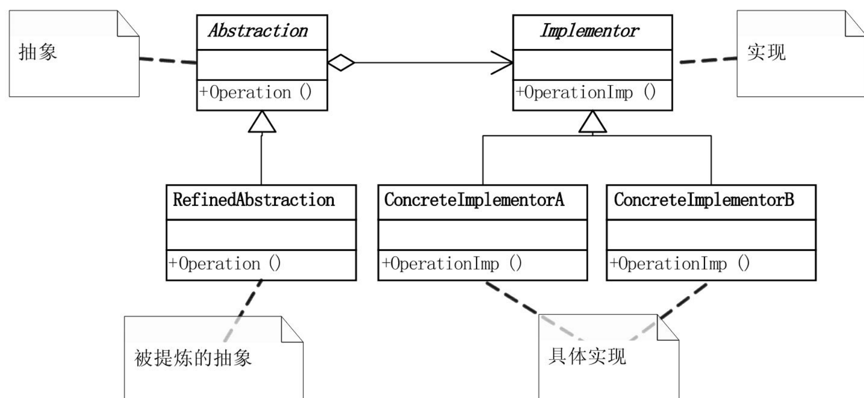


“由于实现的方式有多种，桥接模式的核心意图就是把这些实现独立出来，让它们各自地变化。这就使得每种实现的变化不会影响其他实现，从而达到应对变化的目的。”



22.6 桥接模式基本代码

桥接模式（Bridge）结构图



Implementor类

```
abstract class Implementor
{
    public abstract void Operation();
}
```

ConcreteImplementorA和ConcreteImplementorB等派生类

```
class ConcreteImplementorA : Implementor
{
    public override void Operation()
    {
        Console.WriteLine("具体实现A的方法执行");
    }
}

class ConcreteImplementorB : Implementor
{
    public override void Operation()
    {
        Console.WriteLine("具体实现B的方法执行");
```

```
    }
}
```

Abstraction类

```
class Abstraction
{
    protected Implementor implementor;

    public void  SetImplementor(Implementor implement

    {
        this.implementor = implementor;
    }

    public virtual void Operation()
    {
        implementor.Operation();
    }
}
```

RefinedAbstraction类

```
class RefinedAbstraction : Abstraction
{
    public override void Operation()
    {
        implementor.Operation();
    }
}
```

客户端实现

```
static void Main(string[] args)
{
    Abstraction ab = new RefinedAbstraction();
```

```
            ab.SetImplementor(new ConcreteImplementorA());
            ab.Operation();

            ab.SetImplementor(new ConcreteImplementorB());
            ab.Operation();

            Console.Read();
        }
```

“我觉得桥接模式所说的‘将抽象部分与它的实现部分分离’，还是不好理解，我的理解就是实现系统可能有多角度分类，每一种分类都有可能变化，那么就把这种多角度分离出来让它们独立变化，减少它们之间的耦合。”

“哈，小菜说的和GoF说的不就是一回事吗！只不过你说的更通俗，而人家却更简练而已。也就是说，在发现我们需要多角度去分类实现对象，而只用继承会造成大量的类增加，不能满足开放-封闭原则时，就应该要考虑用桥接模式了。”

“哈，我感觉只要真正深入地理解了设计原则，很多设计模式其实就是原则的应用而已，或许在不知不觉中就在使用设计模式了。”

22.7 我要开发“好”游戏

“说得好，好了，你该干吗就干吗去，我要继续玩游戏了。”大鸟注意力回到了手机上。

“啊，和你说了这么多，我还没来得及看你的新手机呢？”小菜说。

“瞎起什么劲，喜欢自己也去买一个不就得了。”大鸟有些不耐烦了。

“我要是有钱，就一定去买那种有操作系统，把软件与手机分离的智能手机，说不定我还可以自己开发手机游戏呢？”小菜说。

“还是好好打设计模式的基础吧，开发手机游戏不过是一种实际的开发运用而已，有了深厚的功底，学这些具体的新技术又有何难！”

“Yes，Sir！”

第23章　烤羊肉串引来的思考——命令模式

23.1　吃烤羊肉串！

时间：6月23日 17点　　地点：小区门口　　人物：小菜、大鸟

“小菜，肚子饿了，走，我请你吃羊肉串。”

“好呀，小区门口那新疆人烤的就很不错。”

小菜和大鸟来到了小区门口。

“啊，这么多人，都围了十几个。”小菜感叹道。

“现在读大学，进公司，做白领，其实未必有人家烤羊肉串的挣得多。”

“这是两回事，人家也很辛苦呀。”

此时，老板烤的第一批羊肉好了。

“老板，我这有两串。”

“老板，我的是三串不辣的。”

“老板，你怎么给她了，我先付的钱！”

“老板，这串不太熟呀，再烤烤。”

“老板，我老早就等在这里，钱早给你了，你都不给我，我不要了。退钱！”

旁边等着拿肉串的人七嘴八舌地叫开了。场面有些混乱，由于人实在太多，烤羊肉串的老板已经分不清谁是谁，造成分发错误，收钱错误，烤肉质量不过关等等。

“小菜，我看我们还是换一家吧，这里实在太混乱了，过去不远有一家烤肉店是有店面的。”

“嗯，他这样子生意是做不好。咱们去那一家吧。”

时间：6月2日　18点　　地点：烤肉店　　人物：小菜、大鸟、服务员

小菜和大鸟走到了那家烤肉店。

“服务员，我们要十串羊肉串、两串鸡翅、两瓶啤酒。”大鸟根本没有看菜单。

“鸡翅没有了，您点别的烧烤吧。”服务员答道。

“那就来四串牛板筋，烤肉要辣的。”大鸟很轻车熟路。

“大鸟常来这里吃吗？很熟悉嘛！”小菜问道。

“太熟悉了，这年头，单身在外混，哪有不熟悉家门口附近的吃饭的地儿。不然每天晚上的肚皮问题怎么解决？”

“你说，在外面打游击烤羊肉串和这种开门店做烤肉，哪个更赚钱？”小菜问道。

“哈，这很难讲，毕竟各有各的好，在外面打游击，好处是不用租房，不用上税，最多就是交点‘保护费’，但下雨天不行、大白天不行、太晚也不行，一般都是傍晚做几个钟头，顾客也不固定，像刚才那个，由于人多造成混乱，于是就放跑了我们这两条大鱼，其实他的生意是不稳定的。”

“大白天不行？太晚不行？”

“大白天，城管没下班呢，怎能容忍他如此安逸。超过晚上11点，夜深人静，谁还愿意站在路边吃烤肉。但开门店就不一样了，不管什么时间都可以做生意，由于环境相对好，所以固定客户就多，看似好像房租交出去了，但其实由于顾客多，而且是正经做生意，所以最终可以赚到大钱。”

“大鸟研究得很透嘛。”

“其实这门店好过马路游击队，还可以对应一个很重要的设计模式呢！”

“哦，此话怎讲？”

23.2 烧烤摊vs.烧烤店

“你再回忆刚才在我们小区门口烤肉摊看到的情景。”

“因为要吃烤肉的人太多，都希望能最快吃到肉串，烤肉老板一个人，所以有些混乱。”

“还不止这些，老板一个人，来的人一多，他就未必记得住谁交没交过钱，要几串，需不需要放辣等等。”

“是呀，大家都站在那里，没什么事，于是都盯着烤肉去了，哪一串多、哪一串少、哪一串烤得好、哪一串烤得焦看得清清楚楚，于是挑剔也就接踵而至。”

“这其实就是我们在编程中常说的什么？”

“我想想，你是想说‘紧耦合’？”

“哈，不错，不枉我的精心栽培。”

“由于客户和烤羊肉串老板的‘紧耦合’所以使得容易出错，容易混乱，也容易挑剔。”

“说得对，这其实就是‘行为请求者’与‘行为实现者’的紧耦合。我们需要记录哪个人要几串羊肉串，有没有特殊要求（放辣不放辣），付没付过钱，谁先谁后，这其实都相当于对请求做什么？”

“对请求做记录，啊，应该是做日志。”

“很好，那么如果有人需要退回请求，或者要求烤肉重烤，这其实就是？”

“就相当于撤销和重做吧。”

“OK，所以对请求排队或记录请求日志，以及支持可撤销的操作等行为时，‘行为请求者’与‘行为实现者’的紧耦合是不太适合的。你说怎么办？”

“开家门店。”

“哈，这是最终结果，不是这个意思，我们是烤肉请求者，烤肉的师傅是烤肉的实现者，对于开门店来说，我们用得着去看着烤肉的实现过程吗？现实是怎么做的呢？”

“哦，我明白你的意思了，我们不用去认识烤肉者是谁，连他的面都

不用见到，我们只需要给接待我们的服务员说我们要什么就可以了。他可以记录我们的请求，然后再由他去通知烤肉师傅做。”

“而且，由于我们所做的请求，其实也就是我们点肉的订单，上面有很详细的我们的要求，所有的客户都有这一份订单，烤肉师傅可以按先后顺序操作，不会混乱，也不会遗忘了。”

“收钱的时候，也不会多收或少收。”

“优点还不止这里，比如说，”大鸟突然大声叫道，“服务员，我们那十串羊肉串太多了，改成六串就可以了。”

“好的！”服务员答道。

大鸟接着说：“你注意看他接着做了什么？”

“他好像在一个小本子上划了一下，然后去通知烤肉师傅了。”

“对呀，这其实是在做撤销行为的操作。由于有了记录，所以最终算账还是不会错的。”

“对对对，这种利用一个服务员来解耦客户和烤肉师傅的处理好处真的很多。”

“好了，这里有纸和笔，你把刚才的想法写成代码吧？”

“啊，在这？”

“这才叫让编程融入生活。来吧，不写出来，你是不能完全理解的。”

“好吧，我试试看。”

23.3 紧耦合设计

边吃着烤肉串，边写着代码，小菜完成了第一版。

代码结构图



路边烤羊肉串的实现

```
//烤肉串者
public class Barbecuer
{
    //烤羊肉
    public void BakeMutton()
    {
        Console.WriteLine("烤羊肉串!");
    }
    //烤鸡翅
    public void BakeChickenWing()
    {
        Console.WriteLine("烤鸡翅!");
    }
}
```

客户端调用

```
static void Main(string[] args)
{
    Barbecuer boy = new Barbecuer();
    boy.BakeMutton();
    boy.BakeMutton();
    boy.BakeMutton();
    boy.BakeChickenWing();
    boy.BakeMutton();
    boy.BakeMutton();
    boy.BakeChickenWing();

    Console.Read();
}
```

客户端程序与‘烤肉串者’紧耦合，尽管简单，但却极为僵化，有许许多多的隐患

“很好，这就是路边烤肉的对应，如果用户多了，请求多了，就容易乱了。那你再尝试用门店的方式来实现它。”

“我知道一定需要增加服务员类，但怎么做有些不明白。”

“嗯，这里的确是难点，要知道，不管是烤羊肉串，还是烤鸡翅，还是其他烧烤，这些都是‘烤肉串者类’的行为，也就是他的方法，具体怎么做都是由方法内部来实现，我们不用去管它。但是对于‘服务员’类来说，他其实就是根据用户的需要，发个命令，说：‘有人要十个羊肉串，有人要两个鸡翅’，这些都是命令……”

“我明白了，你的意思是，把‘烤肉串者’类当中的方法，分别写成多个命令类，那么它们就可以被‘服务员’来请求了？”

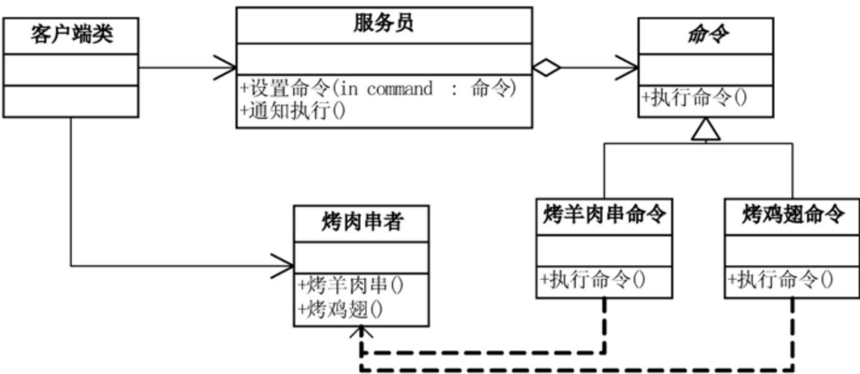
“是的，说得没错，这些命令其实差不多都是同一个样式，于是你就可以泛化出一个抽象类，让‘服务员’只管对抽象的‘命令’发号施令就可以了。具体是什么命令，即是烤什么，由客户来决定吧。”

“我大概明白了。”

23.4 松耦合设计

接着，小菜经过思考，把第二个版本的代码写了出来。

代码结构图



抽象命令类

```
//抽象命令
public abstract class Command
{
    protected Barbecuer receiver;

    public Command(Barbecuer receiver)
    {
        this.receiver = receiver;
    }

    //执行命令
    abstract public void ExcuteCommand();
}
```

抽象命令类，只需要确定‘烤肉串者’是谁

具体命令类

```
//烤羊肉串命令
class BakeMuttonCommand : Command
{
    public BakeMuttonCommand(Barbecuer receiver)
        : base(receiver)
    { }

    public override void ExcuteCommand()
    {
        receiver.BakeMutton();
    }
}

//烤鸡翅命令
class BakeChickenWingCommand : Command
{
    public BakeChickenWingCommand(Barbecuer receiver)
        : base(receiver)
    { }

    public override void ExcuteCommand()
    {
        receiver.BakeChickenWing();
    }
}
```

具体命令类，执行命令时，执行具体的行为

服务员类

```
//服务员
public class Waiter
{
    private Command command;

    //设置订单
    public void SetOrder(Command command)
    {
        this.command = command;
    }
    //通知执行
    public void Notify()
    {
        command.ExcuteCommand();
    }
}
```

服务员类，不用管用户想要什么烤肉，反正都是'命令'，只管记录订单，然后通知'烤肉串者'执行即可

烤肉串者类与之前相同，略。

客户端实现

```
static void Main(string[] args)
{
    //开店前的准备
    Barbecuer boy = new Barbecuer();
    Command bakeMuttonCommand1 = new BakeMuttonCommand(boy);
    Command bakeMuttonCommand2 = new BakeMuttonCommand(boy);
    Command bakeChickenWingCommand1 = new BakeChickenWingCommand(boy);
    Waiter girl = new Waiter();

    //开门营业
    girl.SetOrder(bakeMuttonCommand1);
    girl.Notify();
    girl.SetOrder(bakeMuttonCommand2);
    girl.Notify();
    girl.SetOrder(bakeChickenWingCommand1);
    girl.Notify();

    Console.Read();
}
```

烧烤店事先就找好了烤肉厨师、服务员和烤肉菜单，就等客户上门

服务员根据用户要求，通知厨房开始制作

“大鸟，我这样写如何？”

“很好很好，基本都把代码实现了。但有几个问题，第一，真实的情况其实并不是用户点一个菜，服务员就通知厨房去做一个，那样不科学，应该是点完烧烤后，服务员一次通知制作；第二，如果此时鸡翅没了，不应该是客户来判断是否还有，客户哪知道有没有呀，应该是服务员或烤肉串者来否决这个请求；第三，客户到底点了哪些烧烤或饮料，这是需要记录日志的，以备收费，也包括后期的统计；第四，客户完全有可能因为点的肉串太多而考虑取消一些还没有制作的肉串。这些问题都需要得到解决。”

“你说的这些好像现在都不难办到了。你看着……”

23.5 松耦合后

小菜开始了第三版的代码编写。

服务员类

```
//服务员
public class Waiter
{
    private IList<Command> orders = new List<Command>();

    //设置订单
    public void SetOrder(Command command)
    {
        if (command.ToString() == "命令模式.BakeChickenWingCommand")
        {
            Console.WriteLine("服务员：鸡翅没有了，请点别的烧烤。");
        }
        else
        {
            orders.Add(command);
            Console.WriteLine("增加订单：" + command.ToString() +
                 "  时间：" + DateTime.Now.ToString());
        }
    }

    //取消订单
    public void CancelOrder(Command command)
    {
        orders.Remove(command);
        Console.WriteLine("取消订单：" + command.ToString() +
                 "  时间：" + DateTime.Now.ToString());
    }

    //通知全部执行
    public void Notify()
    {
        foreach (Command cmd in orders)
        {
            cmd.ExcuteCommand();
        }
    }
}
```

增加存放具体命令的容器

在客户提出请求时，对没货的烧烤进行回绝

记录客户所点的烧烤的日志，以备算账收钱

取消订单

根据用户点好的烧烤订单通知厨房制作

客户端代码实现

```
static void Main(string[] args)
{
    //开店前的准备
    Barbecuer boy = new Barbecuer();
    Command bakeMuttonCommand1 = new BakeMuttonCommand(boy);
    Command bakeMuttonCommand2 = new BakeMuttonCommand(boy);
    Command bakeChickenWingCommand1 = new BakeChickenWingCommand(boy);
    Waiter girl = new Waiter();

    //开门营业 顾客点菜
    girl.SetOrder(bakeMuttonCommand1);
    girl.SetOrder(bakeMuttonCommand2);
    girl.SetOrder(bakeChickenWingCommand1);

    //点菜完毕，通知厨房
    girl.Notify();                      订单下好后，一次通知

    Console.Read();
}
```

执行结果：

日志：命令模式.烤羊肉串 时间：200x-xx-xx xx:xx:xx
日志：命令模式.烤羊肉串 时间：200x-xx-xx xx:xx:xx
服务员：鸡翅没有了，请点别的烧烤。
烤羊肉串！
烤羊肉串！

“哈，这就比较完整了。”大鸟满意地点点头。

“你还没有说这是什么设计模式呢。”

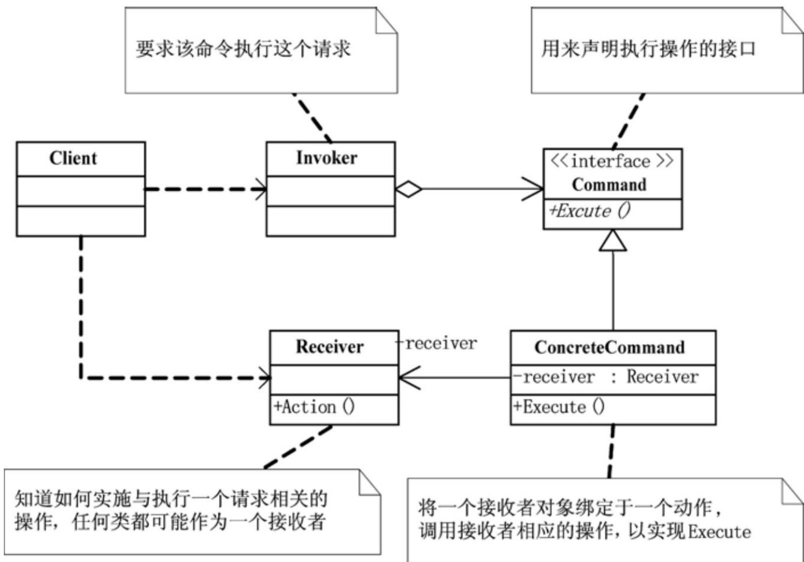
“哈，你猜也应该猜得出，你的那个抽象类叫什么？”

“命令，哦，这就是大名鼎鼎的命令模式呀。”

23.6 命令模式

命令模式（Command），将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。[DP]

命令模式（Command）结构图



Command类，用来声明执行操作的接口。

```
abstract class Command
{
    protected Receiver receiver;

    public Command(Receiver receiver)
    {
        this.receiver = receiver;
    }

    abstract public void Execute();
}
```

ConcreteCommand类，将一个接收者对象绑定于一个动作，调用接

收者相应的操作，以实现Execute。

```
class ConcreteCommand : Command
{
    public ConcreteCommand(Receiver receiver) : base
    { }

    public override void Execute()
    {
        receiver.Action();
    }
}
```

Invoker类，要求该命令执行这个请求。

```
class Invoker
{
    private Command command;

    public void SetCommand(Command command)
    {
        this.command = command;
    }

    public void ExecuteCommand()
    {
        command.Execute();
    }
}
```

Receiver类，知道如何实施与执行一个与请求相关的操作，任何类都可能作为一个接收者。

```
class Receiver
{
    public void Action()
    {
```

```
        Console.WriteLine("执行请求！");
    }
}
```

客户端代码，创建一个具体命令对象并设定它的接收者。

```
static void Main(string[] args)
{
    Receiver r = new Receiver();
    Command c = new ConcreteCommand(r);
    Invoker i = new Invoker();

    i.SetCommand(c);
    i.ExecuteCommand();

    Console.Read();
}
```

23.7　命令模式作用

“来来来，小菜你来总结一下命令模式的优点。”

“我觉得第一，它能较容易地设计一个命令队列；第二，在需要的情况下，可以较容易地将命令记入日志；第三，允许接收请求的一方决定是否要否决请求。”

“还有就是第四，可以容易地实现对请求的撤销和重做；第五，由于加进新的具体命令类不影响其他的类，因此增加新的具体命令类很容易。其实还有最关键的优点就是命令模式把请求一个操作的对象与知道怎么执行一个操作的对象分割开。**[DP]**”大鸟接着总结说。

“但是否是碰到类似情况就一定要实现命令模式呢？”

“这就不一定了，比如命令模式支持撤销/恢复操作功能，但你还不清楚是否需要这个功能时，你要不要实现命令模式？”

“要，万一以后需要就不好办了。”

“其实应该是不要实现。敏捷开发原则告诉我们，不要为代码添加基于猜测的、实际不需要的功能。如果不清楚一个系统是否需要命令模式，一般就不要着急去实现它，事实上，在需要的时候通过重构实现这个模式并不困难，只有在真正需要如撤销/恢复操作等功能时，把原来的代码重构为命令模式才有意义。**[R2P]**”

“明白。这一顿我请客了。”小菜很开心，大声叫了一句，“服务员，埋单。”

“先生，你们一共吃了28元。”服务员递过来一个收费单。

小菜正准备付钱。

“慢！”大鸟按住小菜的手，“不对呀，我们没有吃10串羊肉串，后来改成6串了。应该是24元。”

服务员去查了查账本，回来很抱歉地说，“真是对不起，我们算错了，应该是24元。”

“小菜，你看到了吧，如果不是服务员做了记录，也就是记日志，单就烤肉串的人，哪记得住烤了多少串，后果就是大家都说不清楚了。”

“还是大鸟精明呀。”

第24章　加薪非要老总批？——职责链模式

24.1　老板，我要加薪！

时间：7月2日　20点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“大鸟，你说我现在干满三个月了，马上要办转正手续，我提提加薪的事情好不好？”小菜问道。

“这要看你这三个月做得如何了。”

“我和刚进来的几个同事比较，我觉得我做得很好。公司每每分配的任务，我基本都可以快速完成。有一次，一段程序需要增加一个分支条件，我立刻想到利用反射、工厂等设计模式来处理，经理对我的设计很满意。”

“哦，学以致用，那是最好不过了。那你不妨向你们经理提一提，你现在那点收入确实也有被剥削之嫌。”

“有你这话，我决定了，明天就向经理说。”

“加薪后如何办啊？”

“哦！知道。请你吃炒面。”

“炒面？搞没搞错，我要吃龙虾！”

“好的好的，大龙虾不敢说，小龙虾还是没问题的。”小菜伸了伸舌头笑道。

时间：7月3日　19点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“小菜，是不是该去吃龙虾了？”大鸟下班回来问道。

“嗨！别提了，不让加。”

“嗯？为什么？”

“今天一早，我对经理如实说了我的想法，希望公司能在转正时增加我的工资待遇。经理说这个情况他也了解，并肯定地说，我的工作很认真，我的能力很强。但加薪他做不了主，他帮我向上提一提。而后他去找了人力资源总监，总监说这事他也做不了主，毕竟刚毕业的大学生加薪的先例没有，但总监说，等总经理来后，向总经理提一提这个事。”

“哈，加个薪，流程走了不少呀，后来如何？”

“我从经理那里得到的消息是，总经理不同意加薪，因为现在大学毕业生这么多，随便都能找得到。三个月就想加薪，不合适。”

“这哪和哪呀，大学毕业生多就会贬值呀，这没道理的。加不加薪，还是要看能力，看贡献。”大鸟愤慨地说。

“是呀，我也觉得非常不爽。我们经理说完这话，叫我安心，他会再努力努力。我感觉他其实也是很无奈的。”

“看来你们经理还是很体谅程序员的苦衷。不过你也别抱太大的希望，总经理不同意，基本就没希望了。”

“是呀，这种不重能力，只重资历的公司待着也没劲哦。你说我是不是该考虑换一换？”

“小子，你才毕业呀，刚工作就想跳槽，心态也太不好了吧。”

“嗨，不提了。今天我们学习哪个模式？”

“你把今天你向经理申请，经理没权利，然后向总监上报，总监也没权限，向总经理上报的事，写成代码来看看吧，注意哦，不一定是加薪，还有可能是请假申请等等。”

“哦，难道这也是模式？我试试看。”

24.2 加薪代码初步

“实现这个场景感觉有些难度哦！”

“首先我觉得无论加薪还是请假，都是一种申请。申请就应该有申请类别、申请内容和申请数量。”

```
//申请
class Request
{
    //申请类别
    private string requestType;
    public string RequestType
    {
        get { return requestType; }
        set { requestType = value; }
    }

    //申请内容
    private string requestContent;
    public string RequestContent
    {
        get { return requestContent; }
        set { requestContent = value; }
    }

    //数量
    private int number;
    public int Number
    {
        get { return number; }
        set { number = value; }
    }
}
```

“然后我觉得经理、总监、总经理都是管理者，他们在对‘申请’处理时，需要做出判断，是否有权决策。”

```
//管理者
class Manager
{
    protected string name;
    public Manager(string name)
    {   this.name = name;   }

    //得到结果
    public void GetResult(string managerLevel, Request request)  {
        if (managerLevel == "经理")   {
            if (request.RequestType == "请假" && request.Number <= 2)    {
                Console.WriteLine("{0}:{1} 数量{2} 被批准",
                          name, request.RequestContent, request.Number);
            }  else   {
                Console.WriteLine("{0}:{1} 数量{2} 我无权处理",
                          name, request.RequestContent, request.Number);
            }
        }
        else if (managerLevel == "总监")   {
            if (request.RequestType == "请假" && request.Number <= 5)   {
                Console.WriteLine("{0}:{1} 数量{2} 被批准",
                          name, request.RequestContent, request.Number);
            }  else  {
                Console.WriteLine("{0}:{1} 数量{2} 我无权处理",
                          name, request.RequestContent, request.Number);
            }
        }
        else if (managerLevel == "总经理")  {
            if (request.RequestType == "请假")   {
                Console.WriteLine("{0}:{1} 数量{2} 被批准",
                          name, request.RequestContent, request.Number);
            }  else if (request.RequestType == "加薪" && request.Number <= 500)  {
                Console.WriteLine("{0}:{1} 数量{2} 被批准",
                          name, request.RequestContent, request.Number);
            }  else if (request.RequestType == "加薪" && request.Number > 500)   {
                Console.WriteLine("{0}:{1} 数量{2} 再说吧",
                          name, request.RequestContent, request.Number);
            }
        }
    }
}
```

比较长的方法，多条的分支，这些其实都是代码坏味道

客户端代码如下

```
static void Main(string[] args)
{

    Manager jinli = new Manager("金利");                          三个管理者
    Manager zongjian = new Manager("宗剑");
    Manager zhongjingli = new Manager("钟精励");

    Request request = new Request();
    request.RequestType = "加薪";
    request.RequestContent = "小菜请求加薪";                      小菜请求加薪 1000
    request.Number = 1000;

    jinli.GetResult("经理", request);
    zongjian.GetResult("总监", request);                         不同的级别对此请求做判断和处理
    zhongjingli.GetResult("总经理", request);

    Request request2 = new Request();
    request2.RequestType = "请假";
    request2.RequestContent = "小菜请假";                         小菜请假 3 天
    request2.Number = 3;

    jinli.GetResult("经理", request2);
    zongjian.GetResult("总监", request2);
    zhongjingli.GetResult("总经理", request2);

    Console.Read();
}
```

“其实我自己也觉得写得不怎么好。”小菜说道，“但也不知道如何去处理这类问题。”

“哪里写得不好？”大鸟问道。

“那个‘管理者’类吧，里面的‘结果’方法比较长，加上有太多的分支判断，这其实是非常不好的设计。”

“说得不错，因为你很难讲当中还会不会增加其他的管理类别，比如项目经理、部门经理、人力总监、副总经理等等。那就意味着都需要去更改这个类，这个类承担了太多的责任，这违背了哪些设计原则？”

“类有太多的责任，这违背了单一职责原则，增加新的管理类别，需要修改这个类，违背了开放-封闭原则。”

“说得好，那你觉得应该如何下手去重构它呢？”

“你刚才提到了可能会增加管理类别，那就意味着这里容易变化，我想把这些公司管理者的类别各做成管理者的子类，这就可以利用多态性来化解分支带来的僵化。”

“那如何解决经理无权上报总监，总监无权再上报总经理这样的功能呢？”

“我想让它们之间有一定的关联，把用户的请求传递，直到可以解决

这个请求为止。”

“说得不错，你其实已经说到了一个行为设计模式‘职责链模式’的意图了。”

24.3 职责链模式

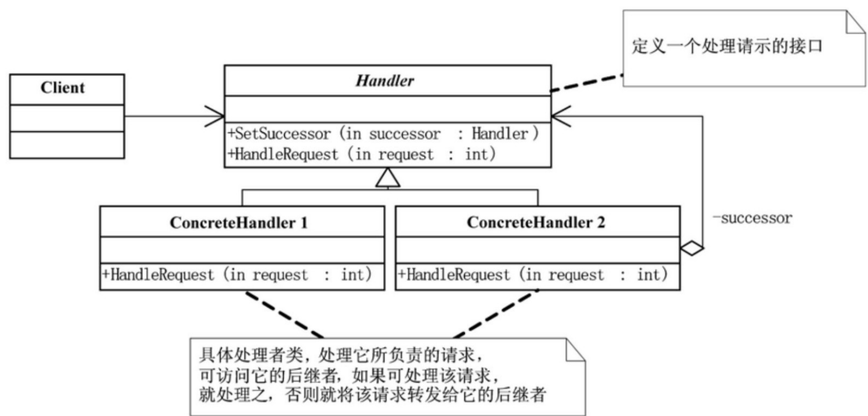
> 职责链模式（Chain of Responsibility）：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这个对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。[DP]

“这里发出这个请求的客户端并不知道这当中的哪一个对象最终处理这个请求，这样系统的更改可以在不影响客户端的情况下动态地重新组织和分配责任。”

“听起来感觉不错哦，但如何做呢？”

“我们来看看结构图。”

职责链模式（**Chain of Responsibility**）结构图



Handler类，定义一个处理请示的接口。

```
abstract class Handler
{
    protected Handler successor;

    public void SetSuccessor(Handler successor)
    {
        this.successor = successor;
    }

    public abstract void HandleRequest(int request);
}
```

设置继任者

处理请求的抽象方法

ConcreteHandler类，具体处理者类，处理它所负责的请求，可访

问它的后继者，如果可处理该请求，就处理之，否则就将该请求转发给它的后继者。

ConcreteHandler1，当请求数在0到10之间则有权处理，否则转到下一位。

```
class ConcreteHandler1 : Handler
{
    public override void HandleRequest(int request)
    {
        if (request >= 0 && request < 10)
        {
            Console.WriteLine("{0} 处理请求 {1}",
              this.GetType().Name, request);
        }
        else if (successor != null)
        {
            successor.HandleRequest(request);
        }
    }
}
```

0 到 10，处理此请求

转移到下一位

ConcreteHandler2，当请求数在10到20之间则有权处理，否则转到下一位。

```
class ConcreteHandler2 : Handler
{
    public override void HandleRequest(int request)
    {
        if (request >= 10 && request < 20)
        {
            Console.WriteLine("{0} 处理请求 {1}",
              this.GetType().Name, request);
        }
        else if (successor != null)
        {
            successor.HandleRequest(request);
        }
    }
}
```

10 到 20，处理此请求

转移到下一位

ConcreteHandler3，当请求数在20到30之间则有权处理，否则转到下一位。

```
class ConcreteHandler3 : Handler
{
    public override void HandleRequest(int request)
    {
        if (request >= 20 && request < 30)
        {
            Console.WriteLine("{0} 处理请求 {1}",
              this.GetType().Name, request);
        }
        else if (successor != null)
        {
            successor.HandleRequest(request);
        }
    }
}
```

20 到 30，处理此请求

转移到下一位

客户端代码，向链上的具体处理者对象提交请求。

```
static void Main(string[] args)
{
    Handler h1 = new ConcreteHandler1();
    Handler h2 = new ConcreteHandler2();
    Handler h3 = new ConcreteHandler3();
    h1.SetSuccessor(h2);
    h2.SetSuccessor(h3);

    int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20 };

    foreach (int request in requests)
    {
        h1.HandleRequest(request);
    }

    Console.Read();
}
```

设置职责链上家与下家

循环给最小处理者提交请求，不同的数额，由不同权限处理者处理

24.4 职责链的好处

“这当中最关键的是当客户提交一个请求时，请求是沿链传递直至有一个ConcreteHandler对象负责处理它。 [DP] ”

“这样做的好处是不是说请求者不用管哪个对象来处理，反正该请求会被处理就对了？”

“是的，这就使得接收者和发送者都没有对方的明确信息，且链中的对象自己也并不知道链的结构。结果是职责链可简化对象的相互连接，它们仅需保持一个指向其后继者的引用，而不需保持它所有的候选接受者的引用 [DP] 。这也就大大降低了耦合度了。”

“我感觉由于是在客户端来定义链的结构，也就是说，我可以随时地增加或修改处理一个请求的结构。增强了给对象指派职责的灵活性 [DP] 。”

“是的，这的确是很灵活，不过也要当心，一个请求极有可能到了链的末端都得不到处理，或者因为没有正确配置而得不到处理，这就很糟糕了。需要事先考虑全面。”

“哈，这就跟现实中邮寄一封信，因地址不对，最终无法送达一样。”

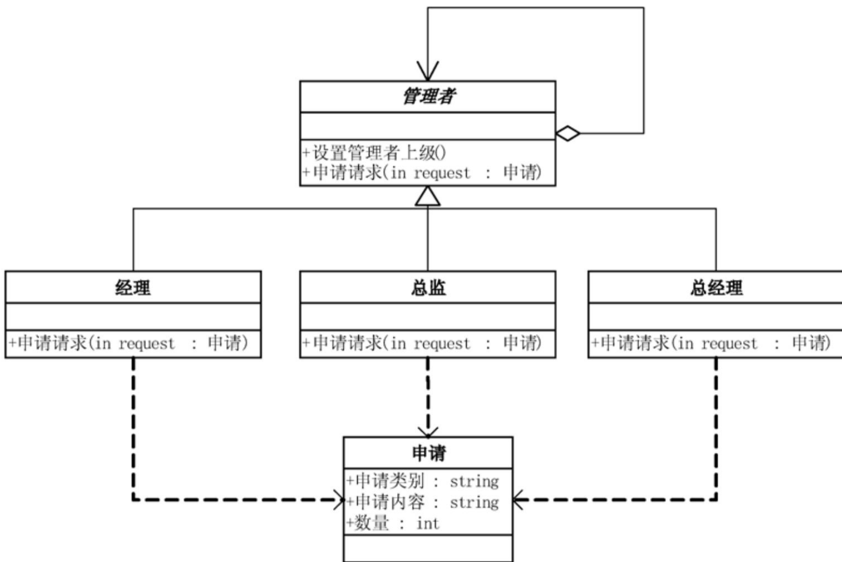
“是的，就是这个意思。就刚才的例子而言，最重要的有两点，一个是你需要事先给每个具体管理者设置他的上司是哪个类，也就是设置后继者。另一点是你需要在每个具体管理者处理请求时，做出判断，是可以处理这个请求，还是必须要‘推卸责任’，转移给后继者去处理。”

“哦，我明白你的意思了，其实就是把我现在写的这个管理者类当中的那些分支，分解到每一个具体的管理者类当中，然后利用事先设置的后继者来实现请求处理的权限问题。”

24.5 加薪代码重构

“是的，所以我们先来改造这个管理者类，此时它将成为抽象的父类了，其实它就是Handler。”

代码结构图




```
//管理者
abstract class Manager
{
    protected string name;
    //管理者的上级
    protected Manager superior;

    public Manager(string name)
    {
        this.name = name;
    }

    //设置管理者的上级
    public void SetSuperior(Manager superior)
    {
        this.superior = superior;
    }
    //申请请求
    abstract public void RequestApplications(Request request);
}
```

关键的方法，设置管理者的上级

“经理类就可以去继承这个‘管理者’类，只需重写‘申请请求’的方法就

可以了。”

```
//经理
class CommonManager : Manager
{
    public CommonManager(string name)
        : base(name)
    { }
    public override void RequestApplications(Request request)
    {

        if (request.RequestType == "请假" && request.Number <= 2)
        {
            Console.WriteLine("{0}:{1} 数量{2} 被批准",
                name, request.RequestContent, request.Number);
        }
        else
        {
            if (superior != null)
                superior.RequestApplications(request);
        }
    }
}
```

经理所能有的权限就是可准许下属两天内的假期

其余的申请都需转到上级

“‘总监’类同样继承‘管理者类’。”

```
//总监
class Majordomo : Manager
{
    public Majordomo(string name)
        : base(name)
    { }
    public override void RequestApplications(Request request)
    {
        if (request.RequestType == "请假" && request.Number <= 5)
        {
            Console.WriteLine("{0}:{1} 数量{2} 被批准",
                name, request.RequestContent, request.Number);
        }
        else
        {
            if (superior != null)
                superior.RequestApplications(request);
        }
    }
}
```

总监所能有的权限就是可准许下属一周内的假期

其余的申请都需转到上级

“‘总经理’的权限就是全部都需要处理。”

```
//总经理
class GeneralManager : Manager
{
    public GeneralManager(string name)
        : base(name)
    { }
    public override void RequestApplications(Request request)
    {

        if (request.RequestType == "请假")
        {
            Console.WriteLine("{0}:{1} 数量{2} 被批准",
                name, request.RequestContent, request.Number);
        }
        else if (request.RequestType == "加薪" && request.Number <= 500)
        {
            Console.WriteLine("{0}:{1} 数量{2} 被批准",
                name, request.RequestContent, request.Number);
        }
        else if (request.RequestType == "加薪" && request.Number > 500)
        {
                Console.WriteLine("{0}:{1} 数量{2} 再说吧",
                    name, request.RequestContent, request.Number);
            }
        }
    }
```

总经理可准许下属任意天的假期

加薪在 500 以内，没有问题

超过 500，就要考虑一下了

“由于我们把你原来的一个‘管理者’类改成了一个抽象类和三个具体类，此时类之间的灵活性就大大增加了，如果我们需要扩展新的管理者类别，只需要增加子类就可以。当然，还有一个关键，那就是客户端如何编写。”

```
static void Main(string[] args)
{
    CommonManager jinli = new CommonManager("金利");
    Majordomo zongjian = new Majordomo("宗剑");
    GeneralManager zhongjingli = new GeneralManager("钟精励");
    jinli.SetSuperior(zongjian);
    zongjian.SetSuperior(zhongjingli);

    Request request = new Request();
    request.RequestType = "请假";
    request.RequestContent = "小菜请假";
    request.Number = 1;
    jinli.RequestApplications(request);

    Request request2 = new Request();
    request2.RequestType = "请假";
    request2.RequestContent = "小菜请假";
    request2.Number = 4;
    jinli.RequestApplications(request2);

    Request request3 = new Request();
    request3.RequestType = "加薪";
    request3.RequestContent = "小菜请求加薪";
    request3.Number = 500;
    jinli.RequestApplications(request3);

    Request request4 = new Request();
    request4.RequestType = "加薪";
    request4.RequestContent = "小菜请求加薪";
    request4.Number = 1000;
    jinli.RequestApplications(request4);

    Console.Read();
}
```

设置上级，完全可以根据实际需求来更改设置

客户端的申请都是由‘经理’发起，但实际谁来决策由具体管理类来处理，客户端不知道

结果显示

```
金利：小菜请假  数量1被批准
宗剑：小菜请假  数量4被批准
钟精励：小菜请求加薪  数量500  被批准
钟精励：小菜请求加薪  数量1000  再说吧
```

“嗯，这的确是很好地解决了原来大量的分支判断造成难维护、灵活性差的问题。”

24.6　加薪成功

正在此时，小菜的手机响了起来。

“喂，”小菜说，“李经理，您好。”

“小蔡呀，是这样，我刚才又去找总经理了，向他也汇报了这一段时间你的工作情况，你的积极表现和非常优秀的编程能力总经理也是非常肯定的。所以他最终答应了你的申请，给你加薪，从下个月开始实施。”小菜的经理在电话那边说道。

“啊，是吗！真的谢谢您，万分感谢。回头我请您吃饭。嗯，好的，好的，OK，拜拜。”小菜脸上笑开了花。

“你这马屁精，干吗不跳槽了？不是说要跳的吗？”

“还跳什么槽呀。我这经理人真的不错，还专门去帮我找总经理谈，太为下属着想了。”

“哈，事实上，他是改变了职责链的结构，跳过了总监直接找总经理处理这事，这也体现了职责链灵活性的一个方面。”

“对的对的，设计模式真是无处不在呀。”小菜笑着说，“走，我请你吃炒面去。”

“谁说是炒面，我讲过的，我要吃的是——龙虾。”

第25章　世界需要和平——中介者模式

25.1　世界需要和平！

时间：7月5日　20点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“大鸟，今晚学习什么模式呢？”小菜问道。

“哦，今天讲中介者模式。”大鸟说。

“中介？就是那个房产或出国留学的中介？”

“哈，是的，就是那个词，中介者模式又叫做调停者模式。其实就是中间人或者调停者的意思。”

“举个例子来说说看。”

“比如最近伊拉克又接连发生了多起爆炸事件，战争带给人类的真是无法弥补的伤痛。”大鸟感慨道，“世界真的需要和平呀。”

“是呀，如果不是伊拉克战争，可能就没这么多事情了。相比较我们可真的太幸福了。”

“再比如巴以问题、伊朗核问题、朝鲜核问题以及各国间的政治外交问题，构成了极为复杂的国际形式。”

“这些和今天要讲的中介者模式有什么关系？”

“你想想看，由于各国之间代表的利益不同，所以矛盾冲突是难免的，但如果有这样一个组织，由各国的代表组成，用来维护国际和平与安全，解决国际间经济、社会、文化和人道主义性质的问题，不就很好吗？”

“啊，你指的就是联合国组织是吧，我明白了，它就是一个调停者、中介者的角色。”

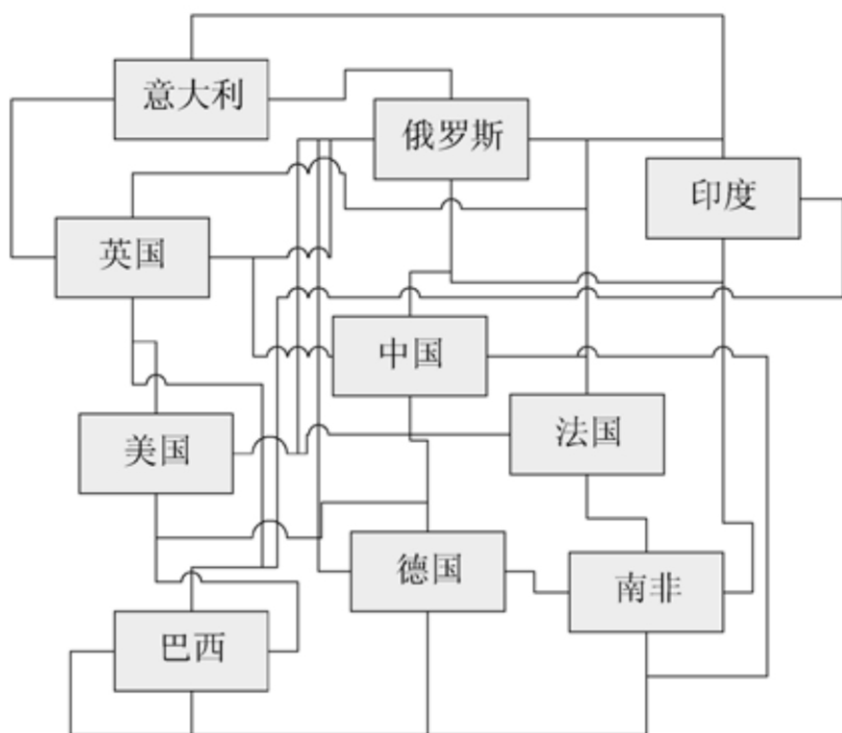
“是的，各国之间关系复杂，战略盟友、战略伙伴、战略对手、利益

相关者等等等等，各国政府都需要投入大量的人力物力在政治、经济、外交方面来搞好这些关系，但不管如何努力，国与国之间的关系还是会随着时间和社会发展而发生改变。在第二次世界大战以前，由于没有这样一个民主中立的协调组织，使得就出现了法西斯联盟，给人类史上造成最大的灾难——二战。而自1945年成立了联合国之后，地球上再没有发生世界范围的战争，可以说，联合国对世界和平的贡献不可估量。”

“啊，原来没有大型战争的原因在这里。那这和我们的软件设计模式又有什么关系呢？”

“你想呀，国与国之间的关系，就类似于不同的对象与对象之间的关系，这就要求对象之间需要知道其他所有对象，尽管将一个系统分割成许多对象通常可以增加其可复用性，但是对象间相互连接的激增又会降低其可复用性了。你知道为什么会这样？”

“我想是因为大量的连接使得一个对象不可能在没有其他对象的支持下工作，系统表现为一个不可分割的整体，所以，对系统的行为进行任何较大的改动就十分困难了。”

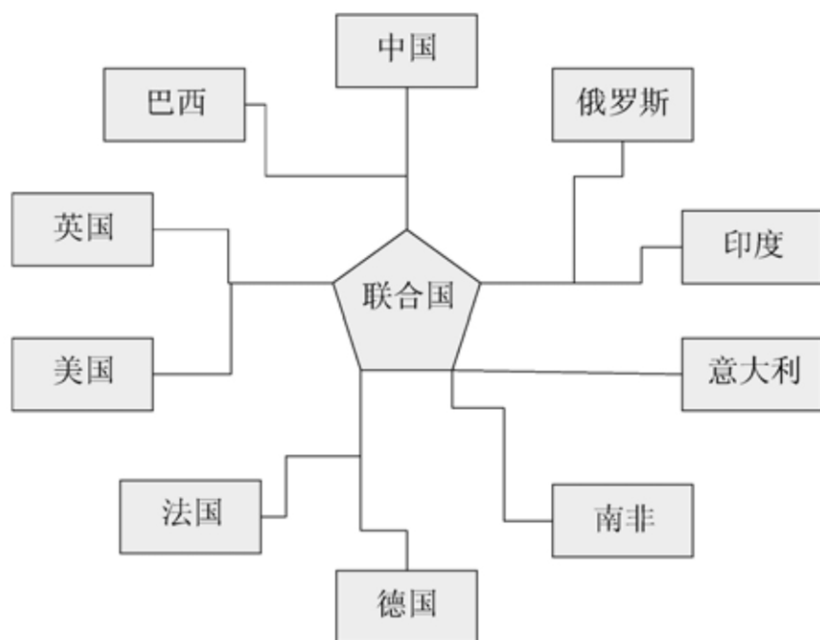


“总结得很好嘛，要解决这样的问题，可以应用什么原则？”

“我记得之前讲过的叫‘迪米特法则’，如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要

调用另一个类的某一个方法的话，可以通过第三者转发这个调用。在这里，你的意思就是说，国与国之间完全可以通过‘联合国’这个中介者来发生关系，而不用直接通信。”

“是呀，通过中介者对象，可以将系统的网状结构变成以中介者为中心的星形结构，每个具体对象不再通过直接的联系与另一个对象发生相互作用，而是通过‘中介者’对象与另一个对象发生相互作用。中介者对象的设计，使得系统的结构不会因为新对象的引入造成大量的修改工作。”



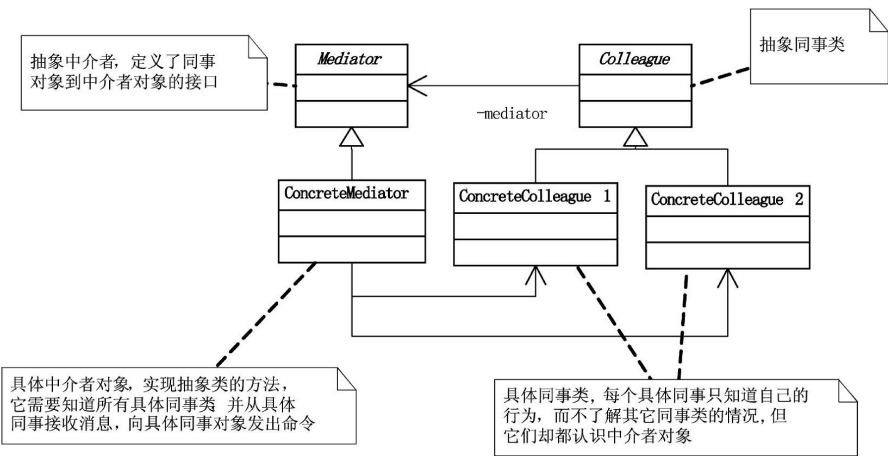
“当时你对我解释‘迪米特法则’的时候，是以我们公司的IT部门的管理为例子的，其实让我一个刚进公司的人去求任何一个不认识的IT部同事帮忙是有困难的，但如果是有IT主管来协调工作，就不至于发生我第一天上班却没有电脑进行工作的局面。IT主管就是一个‘中介者’对象了。”

25.2 中介者模式

“说得好，看来这个模式很容易理解嘛！来，我们看看这个模式的定义。”

中介者模式（**Mediator**），用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。[DP]

中介者模式（**Mediator**）结构图



“Colleague叫做抽象同事类，而ConcreteColleague是具体同事类，每个具体同事只知道自己的行为，而不了解其他同事类的情况，但它们却都认识中介者对象，Mediator是抽象中介者，定义了同事对象到中介者对象的接口，ConcreteMediator是具体中介者对象，实现抽象类的方法，它需要知道所有具体同事类，并从具体同事接收消息，向具体同事对象发出命令。”

Mediator类 抽象中介者类

```
abstract class Mediator
{
    public abstract void Send(string message, Colleague colleague);
}
```

定义一个抽象的发送消息方法，得到同事对象和发送信息

Colleague类 抽象同事类

```
abstract class Colleague
{
    protected Mediator mediator;

    public Colleague(Mediator mediator)
    {
        this.mediator = mediator;
    }
}
```

构造方法，得到中介者对象

ConcreteMediator类 具体中介者类

```
class ConcreteMediator : Mediator
{
    private ConcreteColleague1 colleague1;
    private ConcreteColleague2 colleague2;

    public ConcreteColleague1 Colleague1
    {
        set { colleague1 = value; }
    }

    public ConcreteColleague2 Colleague2
    {
        set { colleague2 = value; }
    }

    public override void Send(string message, Colleague colleague)
    {
        if (colleague == colleague1)
        {
            colleague2.Notify(message);
        }
        else
        {
            colleague1.Notify(message);
        }
    }
}
```

需要了解所有的具体同事对象

重写发送信息的方法，根据对象做出选择判断，通知对象

ConcreteColleague1和ConcreteColleague2等各种同事对象

```
class ConcreteColleague1 : Colleague
{
    public ConcreteColleague1(Mediator mediator)
        : base(mediator)
    {

    }

    public void Send(string message)
    {
```

```
mediator.Send(message,this);
```

```
    }

    public void Notify(string message)
    {
        Console.WriteLine("同事 1 得到信息:"+message);
    }
}

class ConcreteColleague2 : Colleague
{
    public ConcreteColleague2(Mediator mediator):base(mediator)
    {
    }

    public void Send(string message)
    {
        mediator.Send(message, this);
    }

    public void Notify(string message)
    {
        Console.WriteLine("同事 2 得到信息:"+message);
    }
}
```

发送信息时通常是中介者发送出去的

客户端调用

```
static void Main(string[] args)
{
    ConcreteMediator m = new ConcreteMediator();

    ConcreteColleague1 c1 = new ConcreteColleague1(m);
    ConcreteColleague2 c2 = new ConcreteColleague2(m);

    m.Colleague1 = c1;
    m.Colleague2 = c2;

    c1.Send("吃过饭了吗?");
    c2.Send("没有呢，你打算请客?");

    Console.Read();
}
```

让两个具体同事类认识中介者对象

让中介者认识各个具体同事类对象

具体同事类对象的发送信息都是通过中介者转发

“由于有了Mediator，使得ConcreteColleague1和ConcreteColleague2在发送消息和接收信息时其实是通过中介者来完成的，这就减少了它们之间的耦合度了。”大鸟说，“好了，该你来练习了，需求是美国和伊拉克之间的对话都是通过联合国安理会作为中介来完成。”

“哦，这应该没什么大问题，美国和伊拉克都是国家，有一个国家抽象类和两个具体国家类就可以了。但‘联合国’到底是Mediator还是ConcreteMediator呢？”

“这要取决于未来是否有可能扩展中介者对象，比如你觉得联合国除了安理会，还有没有可能有其他机构存在呢？”

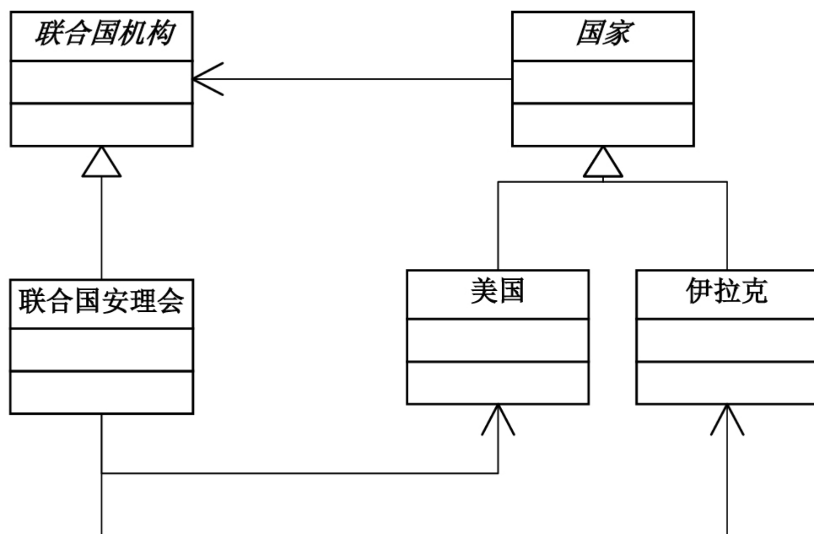
“哈，大鸟你启发我了，联合国的机构还有如国际劳工组织、教科文组织、世界卫生组织、世界贸易组织等等，很多的，所以Mediator应该是‘联合国机构’，而‘安理会’是一个具体的中介者。”

“很好，如果不存在扩展情况，那么Mediator可以与ConcreteMediator合二为一。”大鸟说，“开始吧。”

25.3　安理会做中介

过了近一个小时，小菜才将代码写了出来。

代码结构图



联合国机构类 相当于Mediator类

```
//联合国机构
abstract class UnitedNations
{
    //声明
    public abstract void Declare(string message,
Country colleague);
}
```

国家类 相当于Colleague类

```
//国家
abstract class Country
{
```

```
    protected UnitedNations mediator;

    public Country(UnitedNations mediator)
    {
        this.mediator = mediator;
    }
}
```

美国类 相当于ConcreteColleague1类

```
//美国
class USA : Country
{
    public USA(UnitedNations mediator) : base(media
    { }
    //声明
    public void Declare(string message)
    {
        mediator.Declare(message,this);
    }
    //获得消息
    public void GetMessage(string message)
    {
        Console.WriteLine("美国获得对方信
息：" + message);
    }
}
```

伊拉克类 相当于ConcreteColleague2类

```
//伊拉克
class Iraq : Country
{
    public Iraq(UnitedNations mediator) : base(medi
    { }
    //声明
    public void Declare(string message)
    {
```

```
        mediator.Declare(message,this);
    }
    //获得消息
    public void GetMessage(string message)
    {
        Console.WriteLine("伊拉克获得对方信
息：" + message);
    }
}
```

联合国安理会 相当于ConcreteMediator类

```
//联合国安全理事会
class UnitedNationsSecurityCouncil : UnitedNations
{
    private USA colleague1;
    private Iraq colleague2;
    //美国
    public USA Colleague1
    {  set { colleague1 = value; }  }
    //伊拉克
    public Iraq Colleague2
    {  set { colleague2 = value; }  }

    //声明
    public override void Declare(string message, Country colleague)
    {
        if (colleague == colleague1)
        {
            colleague2.GetMessage(message);
        }
        else
        {
            colleague1.GetMessage(message);
        }
    }
}
```

联合国安理会了解所有的国家，所以拥有美国和伊拉克的对象属性

重写了“声明”方法，实现了两个对象间的通信

客户端调用

```
static void Main(string[] args)
{
    UnitedNationsSecurityCouncil UNSC = new UnitedNat

    USA c1 = new USA(UNSC);
    Iraq c2 = new Iraq(UNSC);

    UNSC.Colleague1 = c1;
```

```
        UNSC.Colleague2 = c2;

        c1.Declare("不准研制核武器，否则要发动战争！");
        c2.Declare("我们没有核武器，也不怕侵略。");

        Console.Read();
    }
```

结果显示

伊拉克获得对方信息：不准研制核武器，否则要发动战争。
美国获得对方信息：我们没有核武器，也不怕侵略。

“小菜呀，你这样的写法和我写的样例代码有什么差别呀，除了类名变量名的不同，没什么区别。这样的代码为何还要写这么长的时间呢？”

“哈，我边写边在思考它是如何做到中介的，其实最关键的问题在于ConcreteMediator这个类必须要知道所有的ConcreteColleague，这好像有些问题？”

“你的想法是什么呢？”

“我觉得尽管这样的设计可以减少了ConcreteColleague类之间的耦合，但这又使得ConcreteMediator责任太多了，如果它出了问题，则整个系统都会有问题了。”

25.4　中介者模式优缺点

“说得好，如果联合国安理会出了问题，当然会对世界都造成影响。所以说，中介者模式很容易在系统中应用，也很容易在系统中误用。当系统出现了‘多对多’交互复杂的对象群时，不要急于使用中介者模式，而要先反思你的系统在设计上是不是合理。你来总结一下中介者模式的优缺点吧。”

“我觉得中介者模式的优点首先是**Mediator**的出现减少了各个**Colleague**的耦合，使得可以独立地改变和复用各个**Colleague**类和**Mediator**，比如任何国家的改变不会影响到其他国家，而只是与安理会发生变化。其次，由于把对象如何协作进行了抽象，将中介作为一个独立的概念并将其封装在一个对象中，这样关注的对象就从对象各自本身的行为转移到它们之间的交互上来，也就是站在一个更宏观的角度去看待系统。比如巴以冲突，本来只能算是国与国之间的矛盾，因此各自的看法可能都比较狭隘，但站在联合国安理会的角度，就可以从全球化、也更客观角度来看待这个问题，在调停和维和上做出贡献。”

“哇，小菜不简单，一个小时的编码，原来你想到这么多，不容易不容易，你说得非常好，用中介者模式的确可以从更宏观的角度来看待问题。那中介者模式的缺点呢？”

“应该就是你刚才提到的，具体中介者类ConcreteMediator可能会因为ConcreteColleague的越来越多，而变得非常复杂，反而不容易维护了。”

“是的，由于**ConcreteMediator**控制了集中化，于是就把交互复杂性变为了中介者的复杂性，这就使得中介者会变得比任何一个**ConcreteColleague**都复杂。事实上，联合国安理会秘书长的工作应该是非常繁忙的，谁叫他就是‘全球最大的官’呢。也正因为此，中介者模式的优点来自集中控制，其缺点也是它，使用时是要考虑清楚哦。”

“啊，这么讲，我感觉中介者模式的应用很少的。”

“小菜，实际上你一直在用它而不自知呀。”大鸟得意地说道。

“哦，有吗？是什么时候？”小菜疑惑着。

“你平时用.NET写的Windows应用程序中的Form或Web网站程序的aspx就是典型的中介者呀。”

“啊，不会吧。这是怎么讲呢？”

“比如计算器程序，它上面有菜单控件、文本控件、多个按钮控件和

一个Form窗体，每个控件之间的通信都是通过谁来完成的？它们之间是否知道对方的存在？”



“哦，我知道了，每个控件的类代码都被封装了，所以它们的实例是不会知道其他控件对象的存在的，比如点击数字按钮要在文本框中显示数字，按照我以前的想法就应该要在Button类中编写给TextBox类实例的Text属性赋值的代码，造成两个类有耦合，这显然是非常不合理的。但实际情况是它们都有事件机制，而事件的执行都是在Form窗体的代码中完成，也就是说所有的控件的交互都是由Form窗体来作中介，操作各个对象，这的确是典型的中介者模式应用。”

“是的，说得很好，”大鸟鼓励道，“中介者模式一般应用于一组对象以定义良好但是复杂的方式进行通信的场合，比如刚才得到的窗体Form对象或Web页面aspx，以及想定制一个分布在多个类中的行为，而又不想生成太多的子类的场合。”

“明白，回到联合国的例子，我相信如果所有的国际安全问题都上升到安理会来解决，世界将不再有战争，世界将会永远和平。”

“让世界充满爱，世界呼唤和平。”

第26章　项目多也别傻做——享元模式

26.1　项目多也别傻做！

时间：7月9日　21点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“小菜，最近一直在忙些什么呢？回家就自个忙开了。”大鸟问道。

“哦，最近有朋友介绍我一些小型的外包项目，是给一些私营业主做网站，我想也不是太难的事情，对自己也是一个很好的编程锻炼，所以最近我都在开发当中。”

“哈，看来小菜有外快赚了，又能锻炼自己的技术，好事呀。”

“现实不是想象的这么简单。刚开始是为一个客户做一个产品展示的网站，我花了一个多星期的时间做好了，也帮他租用了虚拟空间，应该说都很顺利。”

“嗯，产品展示网站，这个应该不难实现。”

“而后，他另外的朋友也希望能做这样网站，我想这有何难，再租用一个空间，然后把之前的代码复制一份上传，就可以了。”

“哦，这好像有点问题，后来呢？”

“实际上却是他们的朋友都希望我来提供这样的网站，但要求就不太一样了，有的人希望是新闻发布形式的，有人希望是博客形式的，也有还是原来的产品图片加说明形式的，而且他们都希望在费用上能大大降低。可是每个网站租用一个空间，费用上降低是不太可能的。我在想如何办呢？”

“他们是不是都是类似的商家客户？要求也就是信息发布、产品展示、博客留言、论坛等功能？”

“是呀，要求差别不大。你说该如何办？”小菜问道。

“你的担心是对的，如果有100家企业来找你做网站，你难道去申请

100个空间，用100个数据库，然后用类似的代码复制100遍，去实现吗？”

“啊，那如果有Bug或是新的需求改动，维护量就太可怕了。”

“先来看看你现在的做法。如果是每个网站一个实例，代码应该是这样的。”

网站类

```
//网站
class WebSite
{
    private string name = "";
    public WebSite(string name)
    {
        this.name = name;
    }
    public void Use()

    {
        Console.WriteLine("网站分类：" + name);
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    WebSite fx = new WebSite("产品展示");
    fx.Use();

    WebSite fy = new WebSite("产品展示");
    fy.Use();

    WebSite fz = new WebSite("产品展示");
    fz.Use();

    WebSite fl = new WebSite("博客");
    fl.Use();
```

```
        WebSite fm = new WebSite("博客");
        fm.Use();

        WebSite fn = new WebSite("博客");
        fn.Use();

        Console.Read();
    }
```

结果显示

```
网站分类：产品展示
网站分类：产品展示
网站分类：产品展示
网站分类：博客
网站分类：博客
网站分类：博客
```

“对的，也就是说，如果要做三个产品展示，三个博客的网站，就需要六个网站类的实例，而其实它们本质上都是一样的代码，如果网站增多，实例也就随着增多，这对服务器的资源浪费得很严重。小菜，你说有什么办法解决这个问题？”

“我不知道，我想过大家的网站共用一套代码，但毕竟是不同的网站，数据都不相同的。”

“我就希望你说出共享代码这句话，为什么不可以呢？比如现在大型的博客网站、电子商务网站，里面每一个博客或商家也可以理解为一个小的网站，但它们是如何做的？”

“啊，我明白了，利用用户ID号的不同，来区分不同的用户，具体数据和模板可以不同，但代码核心和数据库却是共享的。”

“小菜又开窍了，项目多也别傻做呀。你想，首先你的这些企业客户，他们需要的网站结构相似度很高，而且都不是那种高访问量的网站，如果分成多个虚拟空间来处理，相当于一个相同网站的实例对象很多，这是造成服务器的大量资源浪费，当然更实际的其实就是钞票的浪费，如果整合到一个网站中，共享其相关的代码和数据，那么对于硬盘、内存、CPU、数据库空间等服务器资源都可以达成共享，减少服务

器资源，而对于代码，由于是一份实例，维护和扩展都更加容易。”

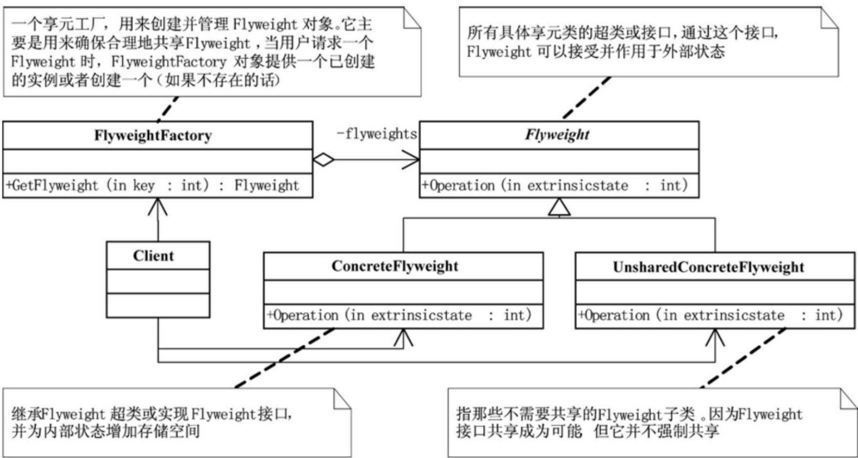
“是的。那如何做到共享一份实例呢？”

26.2 享元模式

“哈，在弄明白如何共享代码之前，我们先来谈谈一个设计模式——享元模式。”

> 享元模式（**Flyweight**），运用共享技术有效地支持大量细粒度的对象。[DP]

享元模式（**flyweight**）结构图



Flyweight类，它是所有具体享元类的超类或接口，通过这个接口，Flyweight可以接受并作用于外部状态。

```
abstract class Flyweight
{
    public abstract void Operation(int extrinsicstate
}
```

ConcreteFlyweight是继承Flyweight超类或实现Flyweight接口，并为内部状态增加存储空间。

```
class ConcreteFlyweight : Flyweight
{
```

```
    public override void Operation(int extrinsicstate
    {
        Console.WriteLine("具体
Flyweight:"+extrinsicstate);
    }
}
```

UnsharedConcreteFlyweight是指那些不需要共享的Flyweight子类。因为Flyweight接口共享成为可能，但它并不强制共享。

```
class UnsharedConcreteFlyweight : Flyweight
{
    public override void Operation(int extrinsicstate
    {
        Console.WriteLine("不共享的具体
Flyweight:" + extrinsicstate);
    }
}
```

FlyweightFactory，是一个享元工厂，用来创建并管理Flyweight对象。它主要是用来确保合理地共享Flyweight，当用户请求一个Flyweight时，FlyweightFactory对象提供一个已创建的实例或者创建一个（如果不存在的话）。

```
class FlyweightFactory
{
    private Hashtable flyweights = new Hashtable();

    public FlyweightFactory()
    {
        flyweights.Add("X", new ConcreteFlyweight());
        flyweights.Add("Y", new ConcreteFlyweight());
        flyweights.Add("Z", new ConcreteFlyweight());
    }

    public Flyweight GetFlyweight(string key)
    {
        return ((Flyweight)flyweights[key]);
    }
}
```


客户端代码

```
static void Main(string[] args)
{
    int extrinsicstate = 22;        // 代码外部状态

    FlyweightFactory f = new FlyweightFactory();

    Flyweight fx = f.GetFlyweight("X");
    fx.Operation(--extrinsicstate);

    Flyweight fy = f.GetFlyweight("Y");
    fy.Operation(--extrinsicstate);

    Flyweight fz = f.GetFlyweight("Z");
    fz.Operation(--extrinsicstate);

    Flyweight uf = new UnsharedConcreteFlyweight();

    uf.Operation(--extrinsicstate);

    Console.Read();
}
```

结果表示

```
具体Flyweight:21
具体Flyweight:20
具体Flyweight:19
不共享的具体Flyweight:18
```

“大鸟，有个问题，”小菜问道，“FlyweightFactory根据客户需求返回早已生成好的对象，但一定要事先生成对象实例吗？”

“问得好，实际上是不一定需要的，完全可以初始化时什么也不做，到需要时，再去判断对象是否为null来决定是否实例化。”

“还有个问题，为什么要有UnsharedConcreteFlyweight的存在呢？”

“这是因为尽管我们大部分时间都需要共享对象来降低内存的损耗，但个别时候也有可能不需要共享的，那么此时的UnsharedConcreteFlyweight子类就有存在的必要了，它可以解决那些不需要共享对象的问题。”

26.3 网站共享代码

“好了，你试着参照这个样例来改写一下帮人做网站的代码。”大鸟接着说。

“哦，好的，那这样的话，网站应该有一个抽象类和一个具体网站类才可以，然后通过网站工厂来产生对象。我马上就去写。”

半小时后，小菜的第二版代码。

网站抽象类

```
abstract class WebSite
{
    public abstract void Use();
}
```

具体网站类

```
class ConcreteWebSite : WebSite
{
    private string name = "";
    public ConcreteWebSite(string name)
    {
        this.name = name;
    }
    public override void Use()
    {
        Console.WriteLine("网站分类：" + name);
    }
}
```

网站工厂类

```
using System.Collections;
//网站工厂
class WebSiteFactory
{
    private Hashtable flyweights = new Hashtable();

    //获得网站分类
    public WebSite GetWebSiteCategory(string key)
    {
        if (!flyweights.ContainsKey(key))
            flyweights.Add(key, new ConcreteWebSite(key));
        return ((WebSite)flyweights[key]);
    }

    //获得网站分类总数
    public int GetWebSiteCount()
    {
        return flyweights.Count;
    }
}
```

判断是否存在这个对象，如果存在，则直接返回，若不存在，则实例化它再返回

得到实例的个数

客户端代码

```
static void Main(string[] args)
{

    WebSiteFactory f = new WebSiteFactory();

    WebSite fx = f.GetWebSiteCategory("产品展示");
    fx.Use();

    WebSite fy = f.GetWebSiteCategory("产品展示");
    fy.Use();

    WebSite fz = f.GetWebSiteCategory("产品展示");
    fz.Use();

    WebSite fl = f.GetWebSiteCategory("博客");
    fl.Use();

    WebSite fm = f.GetWebSiteCategory("博客");
    fm.Use();

    WebSite fn = f.GetWebSiteCategory("博客");
    fn.Use();

    Console.WriteLine("网站分类总数为 {0}", f.GetWebSiteCount());

    Console.Read();
}
```

实例化“产品展示”的“网站”对象

共享上方生成的对象，不再实例化

统计实例化个数，结果应该为 2

结果显示

网站分类：产品展示
网站分类：产品展示
网站分类：产品展示
网站分类：博客
网站分类：博客
网站分类：博客
网站分类总数为 2

“这样写算是基本实现了享元模式的共享对象的目的，也就是说，不管建几个网站，只要是‘产品展示’，都是一样的，只要是‘博客’，也是完全相同的，但这样是有问题的，你给企业建的网站不是一家企业的，它们的数据不会相同，所以至少它们都应该有不同的账号，你怎么办？”

“啊，对的，实际上我这样写没有体现对象间的不同，只体现了它们共享的部分。”

26.4 内部状态与外部状态

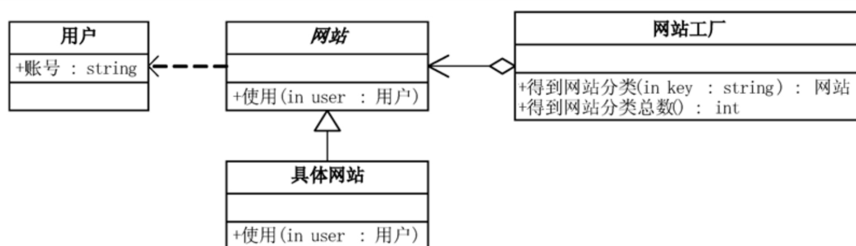
“在享元对象内部并且不会随环境改变而改变的共享部分，可以称为是享元对象的内部状态，而随环境改变而改变的、不可以共享的状态就是外部状态了。事实上，享元模式可以避免大量非常相似类的开销。在程序设计中，有时需要生成大量细粒度的类实例来表示数据。如果能发现这些实例除了几个参数外基本上都是相同的，有时就能够受大幅度地减少需要实例化的类的数量。如果能把那些参数移到类实例的外面，在方法调用时将它们传递进来，就可以通过共享大幅度地减少单个实例的数目。也就是说，享元模式Flyweight执行时所需的状态是有内部的也可能有外部的，内部状态存储于ConcreteFlyweight对象之中，而外部对象则应该考虑由客户端对象存储或计算，当调用Flyweight对象的操作时，将该状态传递给它。”

“那你的意思是说，客户的账号就是外部状态，应该由专门的对象来处理。”

“来，你试试看。”

大约二十分钟后，小菜写出代码第三版。

代码结构图



用户类，用于网站的客户账号，是“网站”类的外部状态

```
//用户
public class User
{
    private string name;
    public User(string name)
    {
        this.name = name;
    }
```

```
    public string Name
    {
        get { return name; }
    }
}
```

网站抽象类

```
abstract class WebSite
{
    public abstract void Use(User user);
}
```

“使用”方法需要传递“用户”对象

具体网站类

```
class ConcreteWebSite : WebSite
{
    private string name = "";
    public ConcreteWebSite(string name)
    {
        this.name = name;
    }

    public override void Use(User user)
    {
        Console.WriteLine("网站分类: " + name + " 用户: " + user.Name);
    }
}
```

实现“Use”方法

网站工厂类

```
//网站工厂
class WebSiteFactory
{
    private Hashtable flyweights = new Hashtable();

    //获得网站分类
    public WebSite GetWebSiteCategory(string key)
    {
        if(!flyweights.ContainsKey(key))
            flyweights.Add(key,
new ConcreteWebSite(key));
        return((WebSite)flyweights[key]);
    }
```

```
    //获得网站分类总数
    public int GetWebSiteCount()
    {
        return flyweights.Count;
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    WebSiteFactory f = new WebSiteFactory();

    WebSite fx = f.GetWebSiteCategory("产品展
示");
    fx.Use(new User("小菜"));

    WebSite fy = f.GetWebSiteCategory("产品展
示");
    fy.Use(new User("大鸟"));

    WebSite fz = f.GetWebSiteCategory("产品展
示");
    fz.Use(new User("娇娇"));

    WebSite fl = f.GetWebSiteCategory("博客");
    fl.Use(new User("老顽童"));

    WebSite fm = f.GetWebSiteCategory("博客");
    fm.Use(new User("桃谷六仙"));

    WebSite fn = f.GetWebSiteCategory("博客");
    fn.Use(new User("南海鳄神"));

    Console.WriteLine("得到网站分类总数为 {0}",
f.GetWebSiteCount());

    Console.Read();
}
```

结果显示 尽管给六个不同用户使用网站，但实际上只有两个网站实例。

网站分类：产品展示 用户：小菜
网站分类：产品展示 用户：大鸟
网站分类：产品展示 用户：娇娇
网站分类：博客 用户：老顽童
网站分类：博客 用户：桃谷六仙
网站分类：博客 用户：南海鳄神
得到网站分类总数为 2

“哈，写得非常好，这样就可以协调内部与外部状态了。由于用了享元模式，哪怕你接手了1000个网站的需求，只要要求相同或类似，你的实际开发代码也就是分类的那几种，对于服务器来说，占用的硬盘空间、内存、CPU资源都是非常少的，这确实是很好的一个方式。”

26.5 享元模式应用

“大鸟，你通过这个例子来讲解享元模式虽然我是理解了，但在现实中什么时候才应该考虑使用享元模式呢？”

“就知道你会问这样的问题，如果一个应用程序使用了大量的对象，而大量的这些对象造成了很大的存储开销时就应该考虑使用；还有就是对象的大多数状态可以外部状态，如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象，此时可以考虑使用享元模式。”

“在实际使用中，享元模式到底能达到什么效果呢？”

“因为用了享元模式，所以有了共享对象，实例总数就大大减少了，如果共享的对象越多，存储节约也就越多，节约量随着共享状态的增多而增大。”

“能具体一些吗？有些什么情况是用到享元模式的？”

“哈，实际上在.NET中，字符串string就是运用了Flyweight模式。举个例子吧。Object.ReferenceEquals（object objA，object objB）方法是用来确定objA与objB是否是相同的实例，返回值为bool值。”

```
string titleA = "大话设计模式";
string titleB = "大话设计模式";
Console.WriteLine(Object.ReferenceEquals(titleA,
titleB));
```

“啊，返回值竟然是True，这两个字符串是相同的实例。”

“试想一下，如果每次创建字符串对象时，都需要创建一个新的字符串对象的话，内存的开销会很大。所以如果第一次创建了字符串对象titleA，下次再创建相同的字符串titleB时只是把它的引用指向‘大话设计模式’，这样就实现了‘大话设计模式’在内存中的共享。”

“哦，原来我一直在使用享元模式呀，我以前都不知道。还有没有其他现实中的应用呢？”

“虽说享元模式更多的时候是一种底层的设计模式，但现实中也是有应用的。比如说休闲游戏开发中，像围棋、五子棋、跳棋等，它们都有大量的棋子对象，你分析一下，它们的内部状态和外部状态各是什么？”

“围棋和五子棋只有黑白两色、跳棋颜色略多一些，但也是不太变化的，所以颜色应该是棋子的内部状态，而各个棋子之间的差别主要就是位置的不同，所以方位坐标应该是棋子的外部状态。”

“对的，像围棋，一盘棋理论上有361个空位可以放棋子，那如果用常规的面向对象方式编程，每盘棋都可能有两三百个棋子对象产生，一台服务器就很难支持更多的玩家玩围棋游戏了，毕竟内存空间还是有限的。如果用了享元模式来处理棋子，那么棋子对象可以减少到只有两个实例，结果……你应该明白的。”

“太了不起了，这的确是非常好地解决了对象的开销问题。”

“在某些情况下，对象的数量可能会太多，从而导致了运行时的资源与性能损耗。那么我们如何去避免大量细粒度的对象，同时又不影响客户程序，是一个值得去思考的问题，享元模式，可以运用共享技术有效地支持大量细粒度的对象。不过，你也别高兴得太早，使用享元模式需要维护一个记录了系统已有的所有享元的列表，而这本身需要耗费资源，另外享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。因此，应当在有足够多的对象实例可供共享时才值得使用享元模式。”

“哦，明白了，像我给人家做网站，如果就两三个人的个人博客，其实是没有必要考虑太多的。但如果是要开发一个可供多人注册的博客网站，那么用共享代码的方式是一个非常好的选择。”

“小菜，说了这么多，你网站赚到钱了是不是该报答一下呀。”

“哈，如果开发完成后客户非常满意，我一定……我一定……”

“一定什么？怎么这么不爽快。”

“我一定送你一个博客账号！”

“啊！！！”

第27章　其实你不懂老板的心——解释器模式

27.1　其实你不懂老板的心

时间：7月12日　20点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“大鸟，今天我们大老板找我谈话。”小菜对大鸟说道。

“哦，大老板找员工谈话，多少有些事情要发生。”大鸟猜测道。

“不知道呀，反正以前我从来没有直接和他面对面说过话，突然他让秘书叫我去他的办公室。我开始多少有些紧张的。”

“他对你说什么了？”

“他问我最近的工作进展情况如何，有没有什么困难，同事相处如何等等。其实也就是简单的调查民情吧。”

“哦，就这么简单？”

“对了，他夸我来着，说‘小蔡，你在公司表现格外出色，要继续好好努力。’”小菜面有得意。

“哈，你原来是想告诉我你们老板夸奖你呀。”大鸟笑道，“你可别太得意了，这句话可以简单地理解为夸奖，也可能有更深层次的含义哦。”

“那还能怎么理解？”

“老板私下对某员工大加夸奖时，多半是‘最近有更多的任务需要你去完成’的意思。”

“啊，不会吧，已经加班够多的了，难道还要加任务？”

“谁叫你平时表现积极呢，年轻人，多做点也没什么关系，积累经验比什么都强，老板赏识你是好事情呀。他还说什么了吗？”

“他还问我另一个叫梅星的同事的情况。我说他工作也很努力。老板

最后评价了句‘梅星是个普通员工。’”小菜说。

“有这样的说法？哦，这个梅星情况不妙了。”大鸟一脸深沉。

“咦，为什么？老板也没说他不好呀，只是说他是普通员工。”

“通常老板说某个员工是普通员工，其实他的意思是说，这个员工不够聪明，工作能力不足。”

“啊，有这种事，我可真是听不懂了。难道老板所说的话，都是有潜台词的？”

“当然，当然。在职场上混，这些都不懂如何干得出名堂！职场菜鸟与老手的一大区别就在于，是否能察言观色，见风使舵，是否听得懂别人尤其是老板上司的弦外之音。”

“大鸟不但在编程技术上有所造诣，连这等为人处世之道也深有研究？”

“略知一二吧，其实自己用点心，这也不算什么难事的。”

“要是有一个翻译机，或解释器就好了，省得每次讲话还需要多动脑筋，多烦呀。”

“小子，你真是块做软件的料，什么问题都想着靠编程解决呀？这种东西要靠感悟的，时间长了，你就会慢慢学会分析的。”大鸟提醒道，“不过，你说到了解释器，我的确是想跟你讲讲解释器模式，它其实就是用来翻译文法句子的。”

“是吗，说来听听。”

27.2 解释器模式

> 解释器模式（**interpreter**），给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。[DP]

“解释器模式需要解决的是，如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题 **[DP]** 。比方说，我们常常会在字符串中搜索匹配的字符或判断一个字符串是否符合我们规定的格式，此时一般我们会用什么技术？”

“是不是正则表达式？”

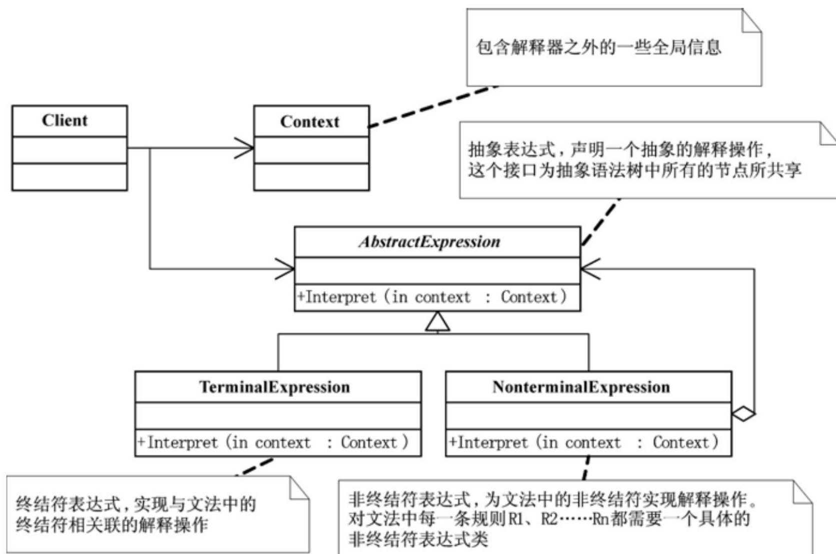
“对，非常好，因为这个匹配字符的需求在软件的很多地方都会使用，而且行为之间都非常类似，过去的做法是针对特定的需求，编写特定的函数，比如判断Email、匹配电话号码等等，与其为每一个特定需求都写一个算法函数，不如使用一种通用的搜索算法来解释执行一个正则表达式，该正则表达式定义了待匹配字符串的集合[DP]。而所谓的解释器模式，正则表达式就是它的一种应用，解释器为正则表达式定义了一个文法，如何表示一个特定的正则表达式，以及如何解释这个正则表达式。”

“我的理解，是不是像IE、Firefox这些浏览器，其实也是在解释HTML文法，将下载到客户端的HTML标记文本转换成网页格式显示到用户？”

“哈，是可以这么说，不过编写一个浏览器的程序，当然要复杂得多。”

“下面我们来看看解释器模式实现的结构图和基本代码。”

解释器模式（**interpreter**）结构图



AbstractExpression（抽象表达式），声明一个抽象的解释操作，这个接口为抽象语法树中所有的节点所共享。

```
abstract class AbstractExpression
{
    public abstract void Interpret(Context context);
}
```

TerminalExpression（终结符表达式），实现与文法中的终结符相关联的解释操作。实现抽象表达式中所要求的接口，主要是一个interpret（）方法。文法中每一个终结符都有一个具体终结表达式与之相对应。

```
class TerminalExpression : AbstractExpression
{
    public override void Interpret(Context context)
    {
        Console.WriteLine("终端解释器");
    }
}
```

NonterminalExpression（非终结符表达式），为文法中的非终结符实现解释操作。对文法中每一条规则R1、R2……Rn都需要一个具体的非终结符表达式类。通过实现抽象表达式的interpret（）方法实现解释操作。解释操作以递归方式调用上面所提到的代表R1、R2……Rn中各个符号的实例变量。

```
class NonterminalExpression : AbstractExpression
{
    public override void Interpret(Context context)
    {
        Console.WriteLine("非终端解释器");
    }
}
```

Context，包含解释器之外的一些全局信息。

```
class Context
{
    private string input;
    public string Input
    {
        get { return input; }
        set { input = value; }
    }

    private string output;
    public string Output
    {
        get { return output; }
        set { output = value; }
    }
}
```

客户端代码，构建表示该文法定义的语言中一个特定的句子的抽象语法树。调用解释操作。

```
static void Main(string[] args)
```

```
{
    Context context = new Context();
    IList<AbstractExpression> list = new List<Abstrac
    list.Add(new TerminalExpression());
    list.Add(new NonterminalExpression());
    list.Add(new TerminalExpression());
    list.Add(new TerminalExpression());

    foreach(AbstractExpression exp in list)
    {
        exp.Interpret(context);
    }

    Console.Read();
}
```

结果显示

终端解释器
非终端解释器
终端解释器
终端解释器

27.3　解释器模式好处

“看起来好像不难，但其实真正做起来应该还是很难的吧。”

“是的，你想，用解释器模式，就如同你开发了一个编程语言或脚本给自己或别人用。这当然是难了。”

“我的理解是，解释器模式就是用‘迷你语言’来表现程序要解决的问题，以迷你语言写成‘迷你程序’来表现具体的问题。”

“嗯，迷你这个词用得很好，就是这样的意思。通常当有一个语言需要解释执行，并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式 **[DP]** 。”

“解释器模式有什么好处呢？”

“用了解释器模式，就意味着可以很容易地改变和扩展文法，因为该模式使用类来表示文法规则，你可使用继承来改变或扩展该文法。也比较容易实现文法，因为定义抽象语法树中各个节点的类的实现大体类似，这些类都易于直接编写 **[DP]** 。”

“除了像正则表达式、浏览器等应用，解释器模式还能用在什么地方呢？”

“只要是可以用语言来描述的，都可以是应用呀。比如针对机器人，如果为了让它走段路还需要去电脑面前调用向前走、左转、右转的方法，那也就太傻了吧。当然应该直接是对它说，‘哥们儿，向前走10步，然后左转90度，再向前走5步。’”

“哈，机器人听得懂‘哥们儿’是什么意思吗？”

“这就看你写的解释器够不够用了，如果你增加这个‘哥们儿’的文法，它就听得懂呀。说白了，解释器模式就是将这样的一句话，转变成实际的命令程序执行而已。而不用解释器模式本来也可以分析，但通过继承抽象表达式的方式，由于依赖倒转原则，使得对文法的扩展和维护都带来了方便。”

“哈，难道说，C#、Java这些高级语言都是用解释器模式的方式开发的？”

“当然不是那么简单了，解释器模式也有不足的，解释器模式为文法中的每一条规则至少定义了一个类，因此包含许多规则的文法可能难以管理和维护。建议当文法非常复杂时，使用其他的技术如语法分析程序

或编译器生成器来处理 **[DP]** ”。

“哦，原来还有语法分析器、编译器生成器这样的东东。”

27.4 音乐解释器

“好了，要真正掌握，还需要练习，我们来做个小型的解释器程序。”

“好呀，程序需求是什么？”

“你以前有没有用过QBASIC？”

“没有，听说那是在VB以前，DOS状态下的编程语言。”

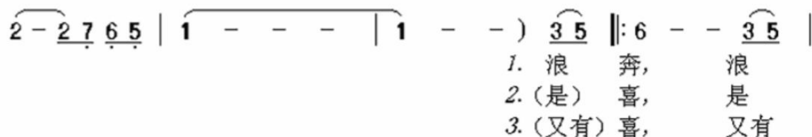
“是的，大鸟我以前就是整天用它来学习写程序的，QBASIC就是早期的BASIC，它当中提供了专门的演奏音乐的语句PLAY，不过由于那会儿多媒体并不像如今这般流行，所以所谓的音乐也仅仅相当于手机中的单音铃声。”大鸟说道。

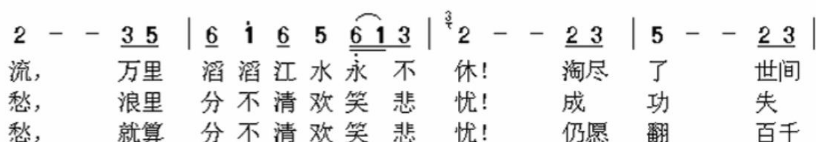
“你一说这个我知道了，我以前的手机里就有编辑铃声的功能，通过输入一些简单的字母数字，就可以让手机发出音乐。我还试着找了些歌谱编了几首流行歌进去呢。”小菜接话道。

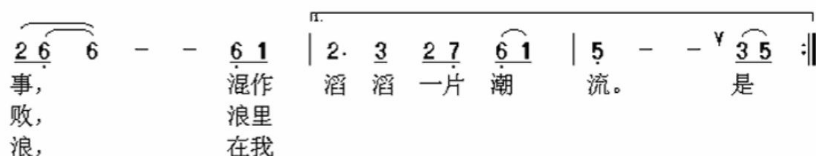
“哈，那就好，你想呀，那就是典型的解释器模式的应用，你用QB或者手机说明书中定义的规则去编写音乐程序，不就是一段文法让QB或手机去翻译成具体的指令来执行吗！”

“我明白了，这就是解释器的应用呀。”

“现在我定义一套规则，和QB的有点类似，但为了简便起见，我做了改动，你就按我定义的规则来编程。我的规则是O 表示音阶‘O 1’表示低音阶，‘O 2’表示中音阶，‘O 3’表示高音阶；‘P ’表示休止符，‘C D E F G A B ’表示‘Do-Re-Mi-Fa-So-La-Ti’；音符长度 1 表示一拍，2表示二拍，0.5表示半拍，0.25表示四分之一拍，以此类推；注意：所有的字母和数字都要用半角空格分开。例如上海滩的歌曲第一句，‘浪奔’，可以写成‘O 2 E 0.5 G 0.5 A 3 ’表示中音开始，演奏的是mi so la。”







“好的，我试试编编看。”

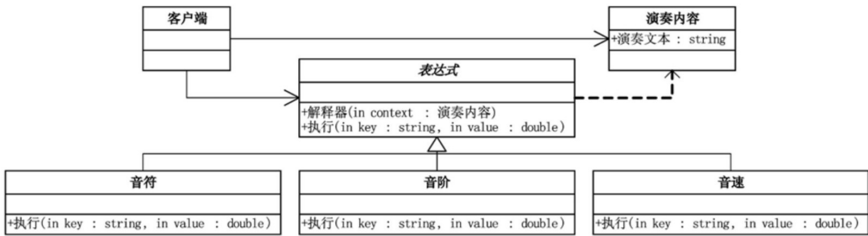
“为了只关注设计模式编程，而不是具体的播放实现，你只需要用控制台根据事先编写的语句解释成简谱就成了。”

“OK！”

27.5 音乐解释器实现

一个小时后，小菜通过了几番改良，给出了答案。

代码结构图



演奏内容类（context）

```
//演奏内容
class PlayContext
{
    //演奏文本
    private string text;
    public string PlayText
    {
        get { return text; }
        set { text = value; }
    }
}
```

表达式类（AbstractExpression）

```
abstract class Expression
{
    //解释器
    public void Interpret(PlayContext context)
    {
        if (context.PlayText.Length == 0)
        {
            return;
        }
        else
        {
            string playKey = context.PlayText.Substring(0, 1);
            context.PlayText = context.PlayText.Substring(2);
            double playValue = Convert.ToDouble(context.PlayText.Substring(0, context.PlayText.
            IndexOf(" ")));
            context.PlayText = context.PlayText.Substring(context.PlayText.IndexOf(" ") +
            1);

            Excute(playKey, playValue);

        }
    }
    //执行
    public abstract void Excute(string key, double value);
}
```

此方法用于将当前的演奏文本第一条命令获得命令字母和其参数值。
例如“O 3 E 0.5 G 0.5 A 3 ”
则 playKey 为 O，而 playValue 为 3

获得 playKey 和 playValue 后将其从演奏文本中移除。
例如“O 3 E 0.5 G 0.5 A 3 ”
变成了“E 0.5 G 0.5 A 3 ”

抽象方法“执行”，不同的文法子类，有不同的执行处理

音符类（TerminalExpression）

```
class Note : Expression
{
    public override void Excute(string key, double value)
    {
        string note = "";
        switch (key)
        {
            case "C":
```

```
                note = "1";
                break;
            case "D":
                note = "2";
                break;
            case "E":
                note = "3";
                break;
            case "F":
                note = "4";
                break;
            case "G":
                note = "5";
                break;
            case "A":
                note = "6";
                break;
            case "B":
                note = "7";
                break;
        }
        Console.Write("{0} ", note);
    }
}
```

表示如果获得的 key 是 C 则演奏 1(do)，如果是 D 则演奏 2（Re）

音阶类（TerminalExpression）

```
class Scale : Expression
{
    public override void Excute(string key, double value)
    {
        string scale = "";
        switch (Convert.ToInt32(value))
        {
            case 1:
                scale = "低音";
                break;
            case 2:
                scale = "中音";
                break;
            case 3:
                scale = "高音";
                break;
        }
        Console.Write("{0} ", scale);
    }
}
```

表示如果获得的 key 是 O 并且 value 是 1 则演奏低音，2 则是中音，3 则是高音

客户端代码

```
static void Main(string[] args)
{
    PlayContext context = new PlayContext();
    //音乐-上海滩
    Console.WriteLine("上海滩：");
    context.PlayText = " O 2 E 0.5 G 0.5 A 3 E 0.5 G 0.5 D 3 E 0.5 G 0.5 A 0.5 O 3 C 1 O
    2 A 0.5 G 1 C 0.5 E 0.5 D 3 ";
    Expression expression=null;
    try
    {
        while (context.PlayText.Length > 0)
        {
            string str = context.PlayText.Substring(0, 1);
            switch (str)
            {
                case "O":
                    expression = new Scale();
                    break;
                case "C":
                case "D":
                case "E":
                case "F":
                case "G":
                case "A":
                case "B":
                case "P":
                    expression = new Note();
                    break;

            }
            expression.Interpret(context);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.Read();
}
```

当首字段是 O 时，则表达式实例化为音阶

当首字母是 CDEFGAB，以及休止符 P 时，则实例化音符

结果显示

上海滩：
中音 3 5 6 3 5 2 3 5 6 高音 1 中音 6 5 1 3 2

“写得非常不错，现在我需要增加一个文法，就是演奏速度，要求是T 代表速度，以毫秒为单位，‘T 1000’表示每节拍一秒，‘T 500’表示每节拍半秒。你如何做？”

“学了设计模式这么久，这点感觉难道还没有，首先加一个表达式的子类叫音速。然后再在客户端的分支判断中增加一个case分支就可以了。”

音速类

```
class Speed : Expression
{
    public override void Excute(string key,
double value)
    {
        string speed;
        if(value < 500)
            speed = "快速";
        else if(value >= 1000)
            speed = "慢速";
        else
            speed = "中速";

        Console.Write("{0} ",speed);
    }
}
```

客户端代码（局部）

```
............                         增加速度的设置
    context.演奏文本 = "T 500 O 2 E 0.5 G 0.5 A 3 E 0.5 G 0.5 D 3 E 0.5 G 0.5 A 0.5 O 3 C
    1 O 2 A 0.5 G 1 C 0.5 E 0.5 D 3 ";
............
switch (str)
         {
            case "O":
                expression = new Scale();
                break;
            case "T":
                expression = new Speed();        增加对 T 的判断
                break;
            case "C":
            case "D":
            case "E":
            case "F":
            case "G":
            case "A":
            case "B":
            case "P":
                expression = new Note();
                break;
         }
............
```

结果显示

上海滩：
中速　中音　3　5　6　3　5　2　3　5　6　高音　1　中音　6　5　1　3　2

“但是小菜，在增加一个文法时，你除了扩展一个类外，还是改动了客户端。”大鸟质疑道。

“哈，这不就是实例化的问题吗，只要在客户端的switch那里应用简单工厂加反射就可做到不改动客户端了。”

“说得好，看来你的确是学明白了，在这里是讲解释器模式，也就不那么追究了，只要知道可以这样重构程序就行。其实这个例子是不能代表解释器模式的全貌的，因为它只有终结符表达式，而没有非终结符表达式的子类，因为如果想真正理解解释器模式，还需要去研究其他的例子。另外这是个控制台的程序，如果我给你钢琴所有按键的声音文件，MP3格式，你可以利用Media　Player控件，写出真实的音乐语言解释器程序吗？”

“你的意思是说，只要我按照简谱编好这样的语句，就可以让电脑模拟钢琴弹奏出来？”

“是的。可以吗？”

“当然是可以，连设计模式都学得会，这点算什么。”

“OK，那我就等你哪天给我写出这样的钢琴模拟程序哦。”

“没问题。”

（注：由于钢琴模拟程序相对复杂，书中篇幅所限，代码不在书中显示，但程序功能有一定趣味性，在随书提供的源代码中有参考代码，有兴趣的读者可下载研究。）

27.6　料事如神

时间：7月16日　20点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

小菜："大鸟，你真是料事如神呀，尽管都不是什么好消息，但两件事都让你猜对了。"

"哦，哪两件事？"

"第一，梅星离职了，说是他辞职，可听说实际上是公司叫他走人的。"

"嗨，所以说呀，好好学习，加强自己的市场竞争力还是非常重要的。"

"第二，就是梅星的工作全部转给我做了，这样我的工作量大大提高，一个人干了两个人的活。"

"哈哈，你不是被老板夸得很开心的吗？现在还开心吗？"

"反正就像你说的，趁着年轻，多做点吧。"

"是的，多做点没坏处的。小菜，你在公司表现格外出色，要继续好好努力哦。"

"我怎么现在听着这话那么别扭，大鸟，你少来，我可不是职场菜鸟了，这种虚情假意的夸奖我可不再上当了。"

"真诚地夸奖你，你反而不信了。真是好心被当作了驴肝肺哦。嗨，做好人难，做好男人更难，做真心的好男人更是难上加难呀。"大鸟深情感慨地说道。

"真心好男人？呕！呕！呕！"小菜大作呕吐状。

两人的脸都笑开了花。

第28章　男人和女人——访问者模式

28.1　男人和女人！

时间：7月18日 21点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“……

男人这本书的内容要比封面吸引人，女人这本书的封面通常是比内容更吸引人。

男人的青春表示一种肤浅，女人的青春标志一种价值。

男人在街上东张西望，被称做心怀不轨；女人在路上左瞅右瞧，被叫做明眸善睐。

男人成功时，背后多半有一个伟大的女人。女人成功时，背后大多有一个不成功的男人。

男人失败时，闷头喝酒，谁也不用劝。女人失败时，眼泪汪汪，谁也劝不了。

男人恋爱时，凡事不懂也要装懂。女人恋爱时，遇事懂也装作不懂。

男人结婚时，感慨道：恋爱游戏终结时，‘有妻徒刑’遥无期。女人结婚时，欣慰曰：爱情长跑路漫漫，婚姻保险保平安。

……”

“小菜在发神经呐，念什么男人、女人、恋爱、结婚乱七八糟的东西？”大鸟问道。

“我在网上看到关于男人与女人的区别讨论，很有点意思，抄了几句念着玩玩。”小菜笑着道，“男人结婚时说是判了‘有妻徒刑’，女人结婚时说是买了爱情保险。你说这滑稽吗？”

“本来吗，男人和女人就是完全不相同的两类人，当然在对待各种问题上会有完全不相同的态度。”

“这样的对比还有很多。你听着，我念给你听……”小菜显得很兴奋。

“行了行了，没有事业的成功，你找出再多的男女差异也找不到女朋友的。还是好好学习吧。”大鸟打断了小菜说话，“今天我们需要聊最后一个模式，叫做访问者模式。”

“哦，那我们上课吧。”小菜不得不停止。

“访问者模式讲的是表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。”大鸟开始如夫子般念叨起来。

“我觉得男女对比这么多的原因主要就是因为人类在性别上就只有男人和女人两类。”小菜意犹未尽。

“小菜！你又打断了我。”大鸟一声大喝，“大鸟很生气，后果很严重。”接着说，“你还要不要学，到底是学访问者模式还是讨论男女关系？”

“最好是混在一起同时学习。”小菜调皮地说道。

“狗屁，这是一回事吗？男人女人与访问者模式有屁的关……”大鸟气得脏话都脱口而出，拿起书本，对着小菜的头上就拍了过去。突然，大鸟手停在了空中，“等等，你刚才说什么？”

“我说什么？我说混在一起学习。”小菜快速躲开。

“前面一句，”大鸟似乎想到了什么。

“前面我说了什么，哦，我是说人类只分为男人和女人，所以才会有这么多的对比。”

“OK，就是这句话了。今天咱们就来讨论讨论这男人与女人的问题。”大鸟把举着书本的手放了下来，微笑着说道。

“大鸟，不学习设计模式了？”轮到小菜疑惑了。

“学呀，不过如你所愿，我们混在一起学习。”

“这之间也有关系？”小菜更加丈二和尚摸不着头脑。

“先不谈模式，你能不能把刚才的那些对比，用控制台的程序写出来，打到屏幕上？”大鸟避而不答。

“这个，应该不难实现。不过有什么意义呢？”

“少来废话，写出来再讲。”

“哦。”

28.2 最简单的编程实现

十分钟后，小菜的程序就写出来了。

```
static void Main(string[] args)
{
    Console.WriteLine("男人成功时，背后多半有一个伟大的女人。");
    Console.WriteLine("女人成功时，背后大多有一个不成功的男人。");
    Console.WriteLine("男人失败时，闷头喝酒，谁也不用劝。");
    Console.WriteLine("女人失败时，眼泪汪汪，谁也劝不了。");
    Console.WriteLine("男人恋爱时，凡事不懂也要装懂。");
    Console.WriteLine("女人恋爱时，遇事懂也装作不懂。");

    Console.Read();
}
```

“小菜呀，这样的代码你也拿得出手？”大鸟讥讽地说道。

“你不是要把那些对比打在屏幕上吗？我做到了呀。”小菜理直气壮。

“但这和打印‘Hello World’有什么区别，难道你是第一天学习编程？”

“那你要我怎么样写才可以呢？”

“你至少要分析一下，这里面有没有类可以提炼，有没有方法可以共享什么的。”

“哦，你是这个意思，早说呀。这里面至少男人和女人应该是两个不同的类，男人和女人应该继承人这样一个抽象类。所谓的成功、失败或恋爱都是指人的一个状态，是一个属性。成功时如何如何，失败时如何如何不过是一种反应。好了，我写得出来了。”小菜开始得意道。

“这还有点面向对象的意思。快点写吧。”

28.3 简单的面向对象实现

半小时后，小菜写出了第二版的程序。

“人”类，是“男人”和“女人”类的抽象类

```
abstract class Person
{
    protected string action;
    public string Action
    {
        get { return action; }
        set { action = value; }
    }
    //得到结论或反应
    public abstract void GetConclusion();
}
```

“男人”类

```
class Man : Person
{
    //得到结论或反应
    public override void GetConclusion()
    {
        if (action == "成功")
        {
            Console.WriteLine("{0}{1}时，背后多半有一个伟大的女人。", this.GetType().Name,
            action);
        }
        else if (action == "失败")
        {
            Console.WriteLine("{0}{1}时，闷头喝酒，谁也不用劝。",this.GetType().Name,action);
        }
        else if (action == "恋爱")
        {
            Console.WriteLine("{0}{1}时，凡事不懂也要装懂。", this.GetType().Name, action);
        }
    }
}
```

this.GetType().Name 是获得当前类的名称，比如这里就是‘男人’

“女人”类

```
class Woman : Person
{
    //得到结论或反应
    public override void GetConclusion()
    {
        if(action == "成功")
        {
          Console.WriteLine("{0}{1}时，背后大多有一
个不成功的男人。",this.GetType().Name,
          action);
        }
        else if(action == "失败")
        {
            Console.WriteLine("{0}{1}时，眼泪汪
汪，谁也劝不了。",this.GetType().Name,action);
        }
        else if(action == "恋爱")
        {
            Console.WriteLine("{0}{1}时，遇事懂也
装作不懂。",this.GetType().Name,action);
        }
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    IList<Person> persons = new List<Person>();

    Person man1 = new Man();
    man1.Action = "成功";
    persons.Add(man1);
    Person woman1 = new Woman();
    woman1.Action = "成功";
    persons.Add(woman1);

    Person man2 = new Man();
    man2.Action = "失败";
    persons.Add(man2);
    Person woman2 = new Woman();
```

```
        woman2.Action = "失败";
        persons.Add(woman2);

        Person man3 = new Man();
        man3.Action = "恋爱";
        persons.Add(man3);
        Person woman3 = new Woman();
        woman3.Action = "恋爱";
        persons.Add(woman3);

        foreach(Person item in persons)
        {
            item.GetConclusion();
        }

        Console.Read();
    }
```

结果显示

```
Man成功时，背后多半有一个伟大的女人。
Woman成功时，背后大多有一个不成功的男人。
Man失败时，闷头喝酒，谁也不用劝。
Woman失败时，眼泪汪汪，谁也劝不了。
Man恋爱时，凡事不懂也要装懂。
Woman恋爱时，遇事懂也装作不懂。
```

“大鸟，现在算是面向对象的编程了吧。”

“粗略看，应该是算，但你不觉得你在‘男人’类与‘女人’类当中的那些if……else……很是碍眼吗？”

“不这样不行呀，反正也不算多。”

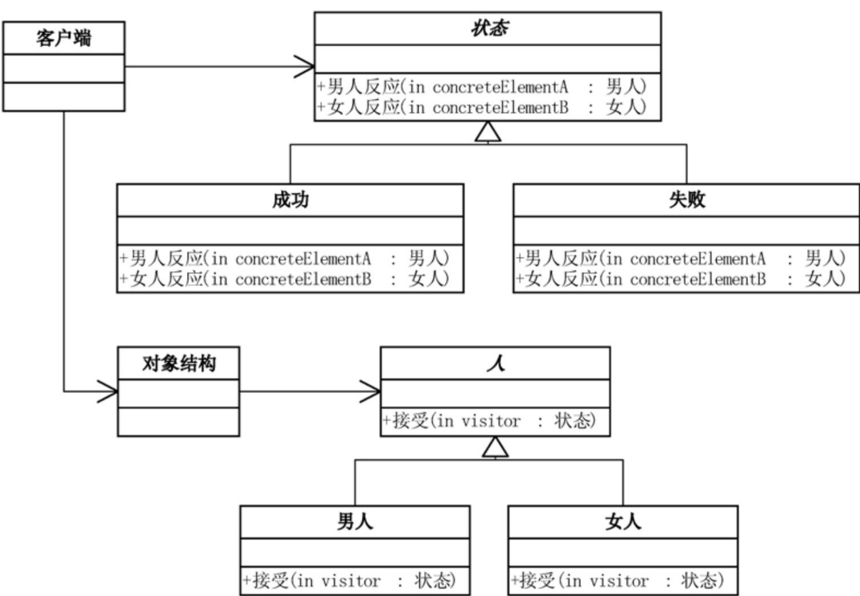
“如果我现在要增加一个‘结婚’的状态，你需要改什么？”

“那这两个类都需要增加分支判断了。”小菜无奈地说，“你说的意思我知道，可是我真的没有办法去处理这些分支，我也想过，把这些状态写成类，可是那又如何处理呢？没办法。”

“哈，办法总是有的。只不过复杂一些。”

28.4 用了模式的实现

大鸟帮助小菜画出了结构图并写出了代码



‘状态’的抽象类和‘人’的抽象类

```
abstract class Action
{
    //得到男人结论或反应
    public abstract void GetManConclusion(Man concreteElementA);
    //得到女人结论或反应
    public abstract void GetWomanConclusion(Woman concreteElementB);
}
abstract class Person
{
    //接受
    public abstract void Accept(Action visitor);
}
```

它是用来获得‘状态’对象的

“这里关键就在于人就只分为男人和女人，这个性别的分类是稳定的，所以可以在状态类中，增加‘男人反应’和‘女人反应’两个方法，方法个数是稳定的，不会很容易的发生变化。而‘人’抽象类中有一个抽象方法‘接受’，它是用来获得‘状态’对象的。每一种具体状态都继承‘状态’抽象类，实现两个反应的方法。”

具体“状态”类

```
//成功
class Success : Action
{
    public override void GetManConclusion(Man concret
    {
        Console.WriteLine("{0}{1}时，背后多半有一个
伟大的女人。",
                                        concreteElementA.GetTy
this.GetType().Name);
    }

    public override void GetWomanConclusion(Woman con
    {
        Console.WriteLine("{0}{1}时，背后大多有一个
不成功的男人。",
                                        concreteElementB.GetTy
this.GetType().Name);
    }
}
//失败
class Failing : Action
{
    //与上面代码类同，省略
}
//恋爱
class Amativeness : Action
{
    //与上面代码类同，省略
}
```

“男人”类和“女人”类

```
//男人
class Man : Person
{
    public override void Accept(Action visitor)
    {
        visitor.GetManConclusion(this);
    }
}

//女人
class Woman : Person
{
    public override void Accept(Action visitor)
    {
        visitor.GetWomanConclusion(this);
    }
}
```

首先在客户程序中将具体状态作为参数传递给“男人”类完成了一次分派，然后“男人”类调用作为参数的“具体状态”中的方法“男人反应”，同时将自己（this）作为参数传递进去。这便完成了第二次分派

“这里需要提一下当中用到一种双分派的技术，首先在客户程序中将具体状态作为参数传递给“男人”类完成了一次分派，然后“男人”类调用作为参数的“具体状态”中的方法“男人反应”，同时将自己（this）作为参数传递进去。这便完成了第二次分派。双分派意味着得到执行的操作决定于请求的种类和两个接收者的类型。‘接受’方法就是一个双分派的操作，它得到执行的操作不仅决定于‘状态’类的具体状态，还决定于它访问的‘人’的类别。”

对象结构类 由于总是需要‘男人’与‘女人’在不同状态的对比，所以我们需要一个‘对象结构’类来针对不同的‘状态’遍历‘男人’与‘女人’，得到不同的反应。

```
//对象结构
class ObjectStructure
{
    private IList<Person> elements = new List<Person>();

    //增加
    public void Attach(Person element)
    {
        elements.Add(element);
    }
    //移除
    public void Detach(Person element)
    {
        elements.Remove(element);
    }
    //查看显示
    public void Display(Action visitor)
    {
        foreach (Person e in elements)
        {
            e.Accept(visitor);
        }
    }
}
```

遍历方法

客户端代码

```
static void Main(string[] args)
{
    ObjectStructure o = new ObjectStructure();
    o.Attach(new Man());
    o.Attach(new Woman());

    //成功时的反应
    Success v1 = new Success();
    o.Display(v1);

    //失败时的反应
    Failing v2 = new Failing();
    o.Display(v2);

    //恋爱时的反应
    Amativeness v3 = new Amativeness();
    o.Display(v3);

    Console.Read();
}
```

在对象结构中加入要对比的“男人”和“女人”

查看在各种状态下，“男人”和“女人”的反应

“这样做到底有什么好处呢？”小菜问道。

“你仔细看看，现在这样做，就意味着，如果我们现在要增加‘结婚’的状态来考查‘男人’和‘女人’的反应。只需要怎么就可以了？”

“哦，我明白你意思了，由于用了双分派，使得我只需要增加一个‘状态’子类，就可以在客户端调用来查看，不需要改动其他任何类的代码。”

“来，写出来试试看。”

结婚状态类

```
class Marriage : Action
{
    public override void GetManConclusion(Man concret
    {
        Console.WriteLine("{0}{1}时，感慨道：恋爱游
戏终结时，‘有妻徒刑’遥无期。",
                    concreteElementA.GetType().
this.GetType().Name);
    }

    public override void GetWomanConclusion(Woman cor
```

```
        {
            Console.WriteLine("{0}{1}时，欣慰曰：爱情长
    跑路漫漫，婚姻保险保平安。",
                                        concreteElementB.GetType().
    this.GetType().Name);
        }
    }
```

客户端代码，增加下面一段代码就可以完成

```
…………
Marriage v4 = new Marriage();
o.Display(v4);
…………
```

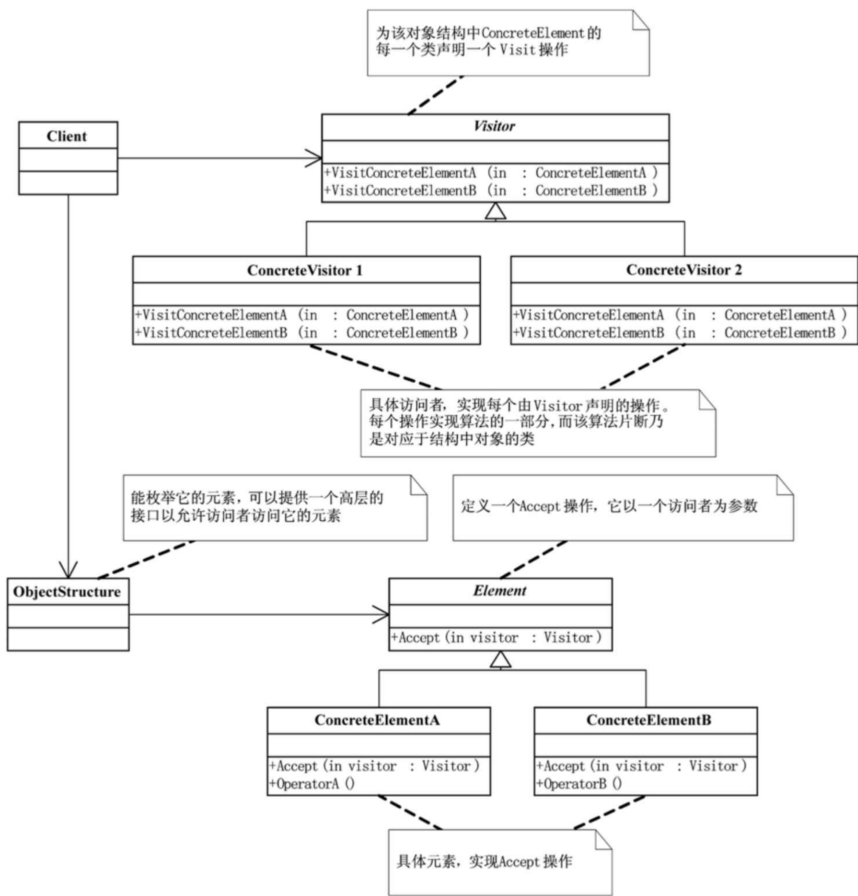
“哈，完美的体现了开放-封闭原则，实在是高呀。这叫什么模式来着？”

“它应该算是GoF中最复杂的一个模式了，叫做访问者模式。”

28.5 访问者模式

> **访问者模式**（**Visitor**），表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。[DP]

访问者模式（**Visitor**）结构图



“在这里，Element就是我们的‘人’类，而ConcreteElementA和ConcreteElementB就是‘男人’和‘女人’，Visitor就是我们写的‘状态’类，具体的ConcreteVisitor就是那些‘成功’、‘失败’、‘恋爱’等等状态。至于ObjectStructure就是‘对象结构’类了。”

“哦，怪不得这幅类图我感觉和刚才写的代码类图几乎可以完全对应。”

“本来我是想直接来谈访问者模式的，但是为什么我突然会愿意和你

聊男人和女人的对比呢，原因就在于你说了一句话：‘男女对比这么多的原因是因为人类在性别上就只有男人和女人两类。’而这也正是访问者模式可以实施的前提。”

“这个前提是什么呢？”

“你想呀，如果人类的性别不止是男和女，而是可有多种性别，那就意味‘状态’类中的抽象方法就不可能稳定了，每加一种类别，就需要在状态类和它的所有下属类中都增加一个方法，这就不符合开放-封闭原则。”

“哦，也就是说，访问者模式适用于数据结构相对稳定的系统？”

“对的，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由地演化。”

“访问者模式的目的是什么？”小菜问道。

“访问者模式的目的是要把处理从数据结构分离出来。很多系统可以按照算法和数据结构分开，如果这样的系统有比较稳定的数据结构，又有易于变化的算法的话，使用访问者模式就是比较合适的，因为访问者模式使得算法操作的增加变得容易。反之，如果这样的系统的数据结构对象易于变化，经常要有新的数据对象增加进来，就不适合使用访问者模式。”

“那其实访问者模式的优点就是增加新的操作很容易，因为增加新的操作就意味着增加一个新的访问者。访问者模式将有关的行为集中到一个访问者对象中。”

“是的，总结得很好。”大鸟接着说，“通常ConcreteVisitor可以单独开发，不必跟ConcreteElementA或ConcreteElementB写在一起。正因为这样，ConcreteVisitor能提高ConcreteElement之间的独立性，如果把一个处理动作设计成ConcreteElementA和ConcreteElementB类的方法，每次想新增“处理”以扩充功能时就得去修改ConcreteElementA和ConcreteElementB了。这也就是你之前写的代码，在‘男人’和‘女人’类中加了对‘成功’、‘失败’等状态的判断，造成处理方法和数据结构的紧耦合。”

“那访问者的缺点其实也就是使增加新的数据结构变得困难了。”

“所以GoF四人中的一个作者就说过：‘大多时候你并不需要访问者模式，但当一旦你需要访问者模式时，那就是真的需要它了。’事实上，我们很难找到数据结构不变化的情况，所以用访问者模式的机会也就不太多了。这也就是为什么你谈到男人女人对比时我很高兴和你讨论的原因，因为人类性别这样的数据结构是不会变化的。”

“哈，看来是我为你找到了一个好的教学样例。”小菜得意道。

“和往常一样，我们需要书写一些基本的代码来巩固我们的学习。有了UML的类图，相信你应该没什么问题了。”

“OK。”

28.6 访问者模式基本代码

Visitor类，为该对象结构中ConcreteElement的每一个类声明一个Visit操作。

```
abstract class Visitor
{
    public abstract void VisitConcreteElementA(Concre

    public abstract void VisitConcreteElementB(Concre
}
```

ConcreteVisitor1和ConcreteVisitor2类，具体访问者，实现每个由Visitor声明的操作。每个操作实现算法的一部分，而该算法片断乃是对应于结构中对象的类。

```
class ConcreteVisitor1 : Visitor
{
    public override void VisitConcreteElementA(Concre
    {
        Console.WriteLine("{0}被{1}访问",
                                 concreteElementA.GetType().
this.GetType().Name);
    }

    public override void VisitConcreteElementB(Concre
    {
        Console.WriteLine("{0}被{1}访问",
                                 concreteElementB.GetType().
this.GetType().Name);
    }
}

class ConcreteVisitor2 : Visitor
{
        //代码与上类类似，省略
}
```

Element类，定义一个Accept操作，它以一个访问者为参数。

```
abstract class Element
{
    public abstract void Accept(Visitor visitor);
}
```

ConcreteElementA和ConcreteElementB类，具体元素，实现Accept操作。

```
class ConcreteElementA : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementA(this);
    }

        public void OperationA()
    { }
}

class ConcreteElementB : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementB(this);
    }

    public void OperationB()
    { }
}
```

充分利用双分派技术，实现处理与数据结构的分离

其他的相关方法

ObjectStructure类，能枚举它的元素，可以提供一个高层的接口以允许访问者访问它的元素。

```
class ObjectStructure
{
    private IList<Element> elements = new List<Elemen

    public void Attach(Element element)
    {
        elements.Add(element);
    }
    public void Detach(Element element)
    {
```

```
        elements.Remove(element);
    }
    public void Accept(Visitor visitor)
    {
        foreach(Element e in elements)
        {
            e.Accept(visitor);
        }
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    ObjectStructure o = new ObjectStructure();
    o.Attach(new ConcreteElementA());
    o.Attach(new ConcreteElementB());

    ConcreteVisitor1 v1 = new ConcreteVisitor1();
    ConcreteVisitor2 v2 = new ConcreteVisitor2();

    o.Accept(v1);
    o.Accept(v2);

    Console.Read();
}
```

28.7　比上不足，比下有余

“啊，访问者模式比较麻烦哦。”

“是的，访问者模式的能力和复杂性是把双刃剑，只有当你真正需要它的时候，才考虑使用它。有很多的程序员为了展示自己的面向对象的能力或是沉迷于模式当中，往往会误用这个模式，所以一定要好好理解它的适用性。”

“哈，大鸟太高估了我们这些菜鸟程序员了。你说得是没错，不过我估计大多数人不去用它的原因绝不是因为怕误用，而是因为它太过于复杂和晦涩，根本不能理解，不熟悉的东西当然就不会想着去应用它了。”

“对，如果不理解，实在是不可能会想到用访问者模式的。”

“不管男人女人，不懂也要装懂的多得是了。”小菜说道，“这其实不能算是男人的专利。”

“你看的那些所谓的男人和女人的对比，都是不准确的，我给个答案吧，男人与女人最大的区别就是，比上不足，比下有余。”

“啊？！”

第29章　OOTV杯超级模式大赛——模式总结

29.1　演讲任务

时间：7月23日　下午　17点　　　地点：小菜办公室　　人物：小菜、公司开发部经理

“小菜，”开发部经理来到小菜的办公桌前，“最近我听说你在工作中，用到了很多设计模式，而且在你们同一项目组中引发了关于设计模式的学习和讨论，反响非常好。明天全公司要开关于如何提高软件质量的研讨会，我希望到时你能做一个关于设计模式的演讲。”

“我？演讲？给全公司？”小菜很惊讶于经理的这个请求，“还是不要了吧，我从来都没上过台给那么多人讲东西。我怕讲不好的。”

“没事，你就像平时和同事交流设计模式那样说说你在开发中的体会和经验就行了。”

“明天，时间太仓促了吧，来不及的。”小菜想法推脱。

“哈，某位大导演在他的近期的电影里不是已经向众人证实，身材是可以靠挤来获得的。同理可证，时间也是可以挤出来的，晚上抓紧一些，一定行的。就这么定了，你好好准备一下。”说完，经理就离开了小菜的办公桌。

“我……”小菜还来不及再拒绝，已经没了机会。

时间：7月23日 晚上 22点　　地点：小菜自己的卧室　　人物：小菜

“讲什么好呢？”小菜使劲地想呀想，却没有头绪，“该死的大鸟，还不回来，不然也可以请教请教他。”

“真困”，小菜睡眼朦胧，趴到了桌上，打起盹来，不一会，进入了梦乡。

29.2　报名参赛

时间：未知　　地点：未知　　人物：很多

“来来来，快来报名了，超级设计模式大赛，每个人都有机会，每个人都能成功，今天你参加比赛给自己一个机会，明天你就成功还社会一个辉煌。来来来……”只见台子上方一很长的条幅，上写着“OOTV杯超级模式大赛海选”，下面一个帅小伙拿着话筒卖力地吆喝着。

“大姐、二姐，我们也去报名参加吧。”工厂三姐妹中最小的简单工厂说道。

“这种选秀比赛，多得去了，很多是骗人的，没意思。”大姐抽象工厂说。

“我觉得我们三姐妹有机会的，毕竟我们从小就是学这个出身。”二姐工厂方法也很有兴趣参赛。

“大姐，去吧，去吧。”简单工厂拉住抽象工厂的手左右摇摆着。

“行了行了，我们去试试，成就成，不成可别乱哭鼻子。”抽象工厂用手指点了一下简单工厂的鼻子，先打上了预防针。

“放心吧，我们都会成功的。”简单工厂很高兴，肯定地说。

三个人果然顺利通过了海选。但在决赛前的选拔中，工厂方法和抽象工厂都晋级，而简单工厂却不幸落选了。

“唔唔唔……唔唔唔……”简单工厂哭着回到了后台。

“小妹，别哭了，到底发生了什么？”工厂方法搂住她，轻声地问道。

“他们说我不符合开放-封闭原则的精神，”简单工厂哽咽地答道，“所以就把我淘汰了。”

“你在对每一次扩展时都要更改工厂类，这就是对修改开放了，当然不符合开闭原则。”抽象工厂说道，“行了，别哭了，讲好了不许哭鼻子的。”

“哇……哇……哇……”被大姐这么一说，简单工厂放声大哭。

“好了，没关系的，这次不行，还会有下次的。”工厂方法拍了拍她

的后背安慰地说。

“二姐，你要好好加油，为我们工厂家族争光。”简单工厂握着工厂方法的手说，然后对着抽象工厂说，“大姐，哼，你老是说风凉话，祝你早日淘汰。”边说着，简单工厂却破涕为笑了。

“你这小妮子，敢咒我。看我不……”抽象工厂脸上带笑，嘴里骂着，举起手欲拍过去。

“二姐救我，大姐饶命。”简单工厂躲到工厂方法后面叫道。

三姐妹追逐打闹着，落选的情绪一扫而空。

29.3　超模大赛开幕式

“各位领导、各位来宾、观众朋友们，第一届OOTV杯超级设计模式大赛正式开始。”漂亮的主持人GOF出现在台前，话音刚落，现场顿时响起激昂的音乐热烈的掌声。

“首先我们来介绍一下到场的嘉宾。第一位是我们OOTV创始人，面向对象先生。”只见前排一位40多岁的中年人站了起来，向后排的观众挥手。

工厂方法在后台对着抽象工厂说：“没想到面向对象这么年轻，40岁就功成名就了。”

“是呀，他从小是靠做simula服装开始创业的，后来做Smalltalk的生意开始发扬光大，但最终让他成功的是Java，我觉得他也就是运气好。”抽象工厂解释说。

“运气也是要给有准备头脑的人，前二十年，做C品牌服装生意的人多得是了，结构化编程就象神一样的被顶礼膜拜，只有面向对象能坚持OO理念，事实证明，OO被越来越多的人认同。这可不是运气。”

“结构化编程那确实是有些老了，时代不同了，老的偶像要渐渐退出，新的偶像再站出来。现在是面向对象的时代，当然如日中天，再过几年就不一定是他了。”抽象工厂相对悲观。

“面向对象不是一直声称自己‘永远二十五岁’吗？”工厂方法双手抱拳放在胸前，坚定地说，“我看好他，他是我永远的偶像。”

“到场来宾还有，抽象先生、封装先生、继承女士、多态女士。他们也是我们这次比赛的策划、导演和监制，掌声欢迎。”主持人*GOF*说道。

工厂方法说：“啊，这些明星，平时看都看不见的，真想为他们尖叫。”

抽象工厂说：“我最大的愿望就是能得到抽象先生的签名，看来极有可能梦想成真了。”

两姐妹在那里入迷地望着前台，自说自话着。

“现在介绍本次大赛的评委，单一职责先生、开放封闭先生、依赖倒转先生、里氏代换女士、合成聚合复用女士、迪米特先生。”主持人GOF说道。

“那个叫开放封闭的家伙就是提出淘汰小妹的人。”抽象工厂对工厂方法说。

“嘘，小点声，他们可就是我们的评委，我们的命运由他们决定的。”工厂方法把食指放在嘴边小声地说。

“下面有请面向对象先生发表演讲。”主持人*GOF*说道。

面向对象大步流星地走上了台前，没有任何稿子，语音洪亮地开始了发言。

“各位来宾，电视机前的朋友们，大家好！（鼓掌）

感谢大家来为OOTV的超级设计模式大赛捧场。OO从诞生到现在，经历了风风雨雨，我面向对象能有今天真的也非常的不容易。就着这机会，我来谈谈面向对象的由来和举办此次设计模式大赛的目的。

软件开发思想经过了几十年的发展。最早的机器语言编程，程序员一直在内存和外存容量的苛刻限制下‘艰苦’劳作。尽管如此，当时程序员还是创造了许多令人惊奇的工程软件。后来，高性能的计算机越来越普及，它们拥有较多的内外存空间，编程也发展到一个较高的层次，不再对任一细节都斤斤计较，于是出现了各种高级语言，软件编程开始进入了全面开花的时代。

刚开始的高级语言编写，大多是面条式的代码，随着代码的复杂化，这会造成代码极度混乱。随着软件业的发展，面条式的代码是越来越不适应发展的需要，此时出现了结构化编程，即面向过程式的开发，这种方式把代码分割成了多个模块，增强了代码的复用性，方便了调试和修改，但是结构也相对复杂一些。面向过程的开发，把需求理解成一条一条的业务流程，开发前总是喜欢问用户‘你的业务流程是什么样的？’，然后他们分析这些流程，把这些流程交织组合在一起，然后再划分成一个又一个的功能模块，最终通过一个又一个的函数，实现了需求。这对于一个小型的软件来说，或许是最直接最简捷的做法。

而问题也就出在了这里。随着软件的不断复杂化，这样的做法有很大的弊端。面向过程关注业务流程，但无论多么努力工作，分析做得如何好，也是永远无法从用户那里获得所有的需求的，而业务流程却是需求中最可能变化的地方，业务流程的制定需要受到很多条件的限制，甚至程序的效率、运行方式都会反过来影响业务流程。有时候用户也会为了更好地实现商业目的，主动地改变业务流程，并且一个流程的变化经常会带来一系列的变化。这就使得按照业务流程设计的程序经常面临变化。今天请假可能就只需要打声招呼就行了，明天请假就需要多个级别管理者审批才可以。流程的易变性，使得把流程看得很重，并不能适应变化。

面向过程通过划分功能模块，通过函数相互间的调用来实现，但需求变化时，就需要更改函数。而你改动的函数有多少的地方在调用它，关联多少数据，这是很不容易弄得清楚的地方。或许开发者本人弄得清楚，但下一位维护代码者是否也了解所有的函数间的彼此调用关系呢？

函数的修改极有可能引起不必要的Bug的出现，维护和调试中所耗费的大多数时间不是花在修改Bug上，而是花在寻找Bug上，弄清如何避免在修改代码时导致不良副作用上了。种种迹象都表明，面向过程的开发也不能适应软件的发展。

与其抱怨需求总是变化，不如改变开发过程，从而更有效地应对变化。面向对象的编程方式诞生，就是为解决变化带来的问题。

面向对象关注的是对象，对象的优点在于，可以定义自己负责的事物，做要求它自己做的事情。对象应该自己负责自己，而且应该清楚地定义责任。

面向对象的开发者，把需求理解成一个一个的对象，他们喜欢问用户‘这个东西叫做什么，他从哪里来，他能做什么事情？’，然后他们制造这些对象，让这些对象互相调用，符合了业务需要。

需求变化是必然的，那么尽管无法预测会发生什么变化，但是通常可以预测哪里会发生变化。面向对象的优点之一，就是可以封装这些变化区域，从而更容易地将代码与变化产生的影响隔离开来。代码可以设计得使需求的变化不至于产生太大的影响。代码可以逐步演进，新代码可以影响较少地加入。

显然，对象比流程更加稳定，也更加封闭。业务流程从表面上看只有一个入口、一个出口，但是实际上，流程的每一步都可能改变某个数据的内容、改变某个设备的状态，对外界产生影响。而对象则是完全通过接口与外界联系，接口内部的事情与外界无关。

当然，有了面向对象的方式，问题的解决看上去不再这么直截了当，需要首先开发业务对象，然后才能实现业务流程。随着面向对象编程方式的发展，又出现了设计模式、ORM、以及不计其数的工具、框架。软件为什么会越来越复杂呢？其实这不是软件本身的原因，而是因为软件需要解决的需求越来越复杂了。

面向过程设计开发相对容易，但不容易应对变化。面向对象设计开发困难，但却能更好的应对千变万化的世界，所以现代的软件需要面向对象的设计和开发。

（鼓掌）

设计模式是面向对象技术的最新进展之一。由于面向对象设计的复杂性，所以我们都希望能做出应对变化，提高复用的设计方案，而设计模式就能帮助我们做到这样的结果。通过复用已经公认的设计，我们能够在解决问题时避免前人所犯的种种错误，可以从学习他人的经验中获益，用不着为那些总是会重复出现的问题再次设计解决方案。显然，设计模式有助于提高我们的思考层次。让我们能站在山顶而不是山脚，也就是更高的高度来俯视我们的设计。

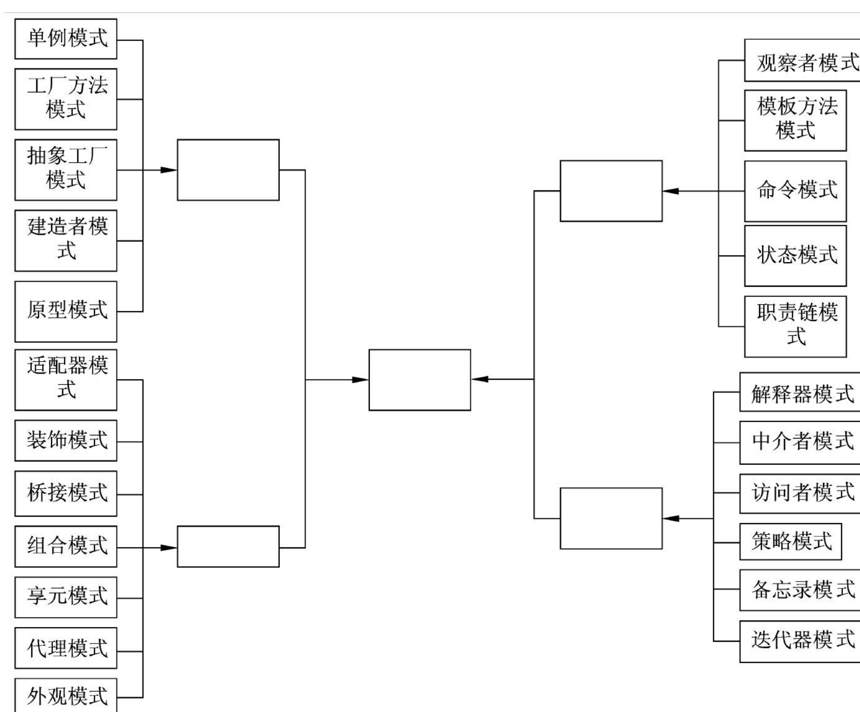
如今，好的设计模式越来越多，但了解他们的人却依然很少，我们OOTV举办设计模式大赛的目的一方面是为了评选出最优秀的设计模式，另一方面也是希望让更多的人了解她们，认识她们，让她们成为明星，让她们可以为您的工作服务。

祝愿本届大赛圆满成功。谢谢大家！[DPE]”（鼓掌）

正在此时，突然一个人双手举着一块牌子冲上了讲台，纸牌上写着“Service-Oriented Architecture（面向服务的体系架构SOA）”，口中大声且反复地说道：“抵制Object-Oriented，推广Service-Oriented，OO已成往事，SOA代表未来。”

这突如其来的变化，让全场哗然，很多人都交头接耳，说着关于SOA与OO的关系。只有面向对象先生依然站在讲台上，微笑不语，显然久经风雨的他对于这种事早已见怪不怪。保安迅速带着此人离开了会场。会场渐渐又恢复了安静。

“下面宣布一下比赛规则。”*GOF*的声音再次响起，“本次大赛根据模式的特点，设置了三个类别，分别是创建型模式、结构型模式和行为型模式，但由于有11位选择了行为型模式，人数过多，所以行为型模式又分为了两组。也就是说，我们将选手共分为了四组，所有的选手都将首先参加分组比赛，每组第一名将参加我们最终设计模式冠军的争夺。选手的分组情况，请看大屏幕。”



“下面我们就有请，单一职责先生代表评委宣誓。”

此时，几位评委站了起来，单一职责先生拿起事先写好的稿子，缓慢地说道：“我代表本届大赛全体评委和工作人员宣誓：恪守职业道德，遵守竞赛规则。严格执法，公正裁判，努力为参赛选手提供良好的比赛氛围和高效优质服务，维护公正的评委信誉。为保证大会的圆满成功，做出我们应有的贡献！宣誓人：单一职责。”

“宣誓人：开放封闭”

“宣誓人：依赖倒转”

……

“下面有请策略模式小姐代表参赛选手宣誓。”

“为了展示面向对象的优点和思想，为了编程的光荣和团队的荣誉，我代表我们全体参赛选手，将弘扬‘可维护、可扩展、可复用、灵活性好’的OO精神，严格遵守赛事活动的各项安排，遵守比赛规则和赛场纪律，尊重对手，团结协作，顽强拼搏，赛出风格，赛出水平，胜不骄，败不馁，尊重裁判，尊重对方，尊重观众。并预祝大赛圆满成功。”

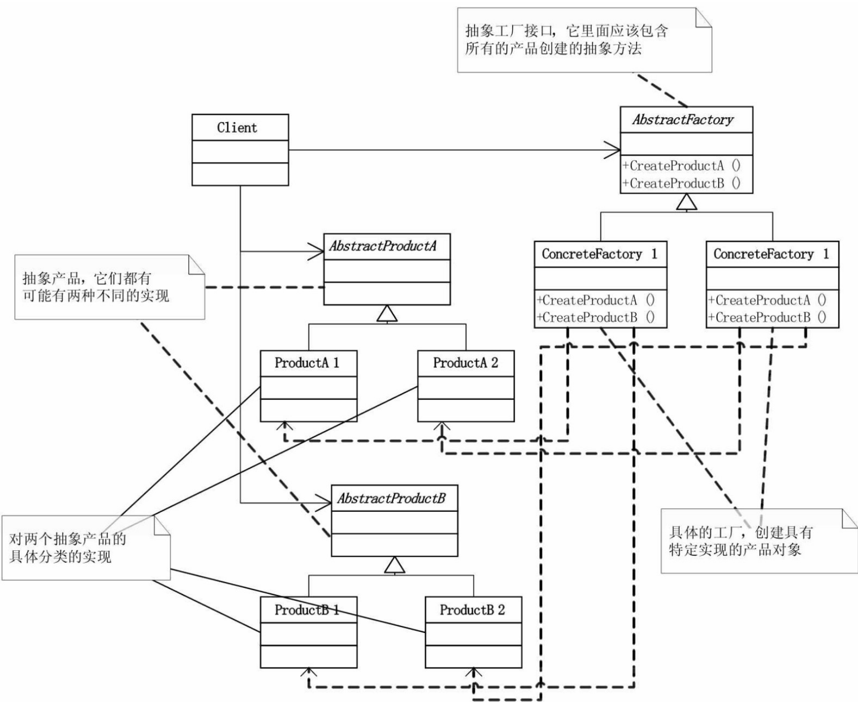
29.4 创建型模式比赛

“现在比赛正式开始，有请第一组参赛选手入场，并进行综合形象展示。”

“第一组创建型选手，他们身穿的是Java正装进行展示。”

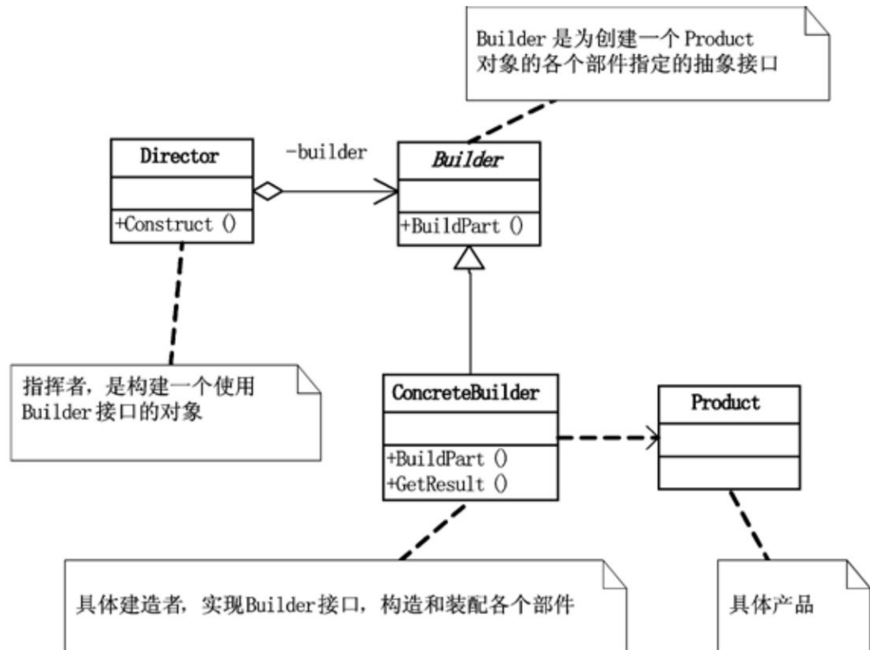
“1号选手，抽象工厂小姐，她的口号是提供一个创建一系列或相关依赖对象的接口，而无需指定它们具体的类。 **[DP]** ”

1号选手 抽象工厂（Abstract Factory）



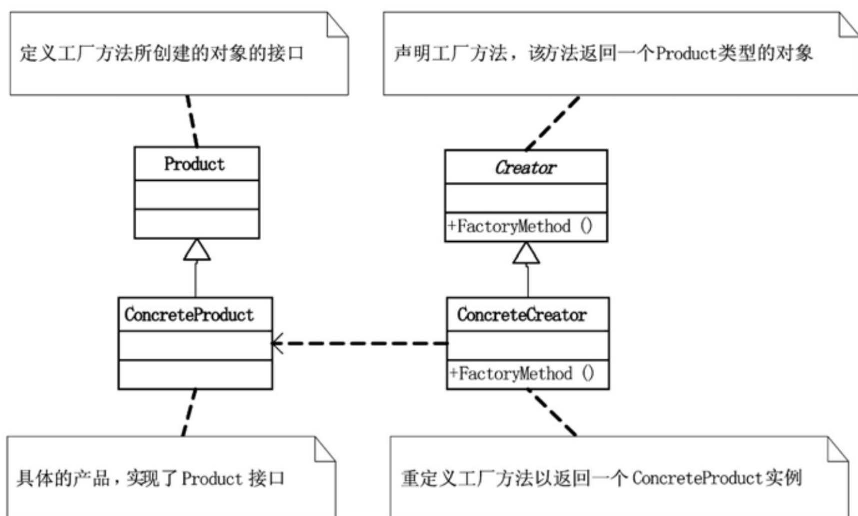
“2号选手，建造者小姐，她的口号是将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。 **[DP]** ”

2号选手 建造者（Bulider）



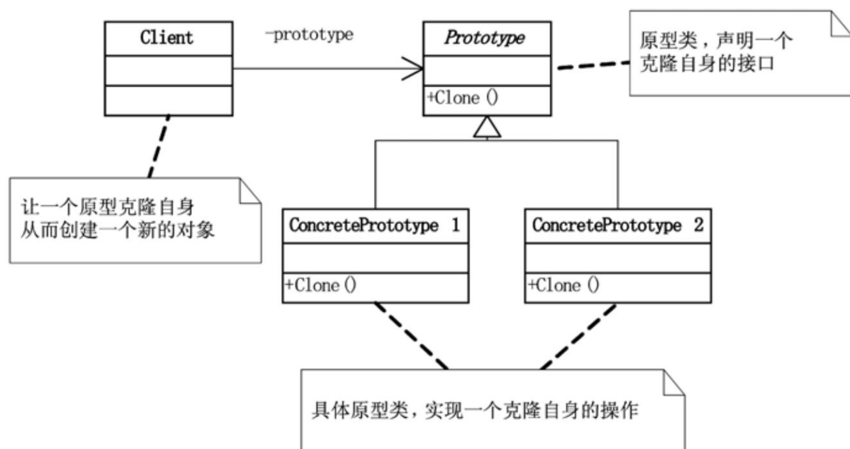
“3号选手工厂方法小姐向我们走来，她声称定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂模式使一个类的实例化延迟到其子类。**[DP]**”

3号选手 工厂方法（**Factory Method**）



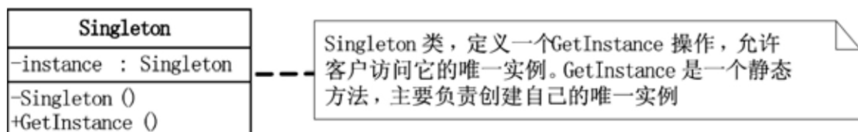
“4号选手是原型小姐，她的意图是用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。**[DP]**”

4号选手 原型（**Prototype**）



“5号选手出场，单例小姐，她提倡简捷就是美，保证一个类仅有一个实例，并提供一个访问它的全局访问点。**[DP]**”

5号选手 单例（**Singleton**）



此时只见场下一帮Fans开始热闹起来。

简单工厂带领着抽象工厂和工厂方法的粉丝们开始齐唱，“咱们工厂有力量，嗨！咱们工厂有力量！每天每日工作忙，嗨！每天每日工作忙，……哎！嗨！哎！嗨！为了咱程序员彻底解放！”

“单例单例，你最美丽，一人创建，全家获益。”单例的Fans同样不甘示弱地喊着口号。

观众席中还有两位先生安静地坐在那里，小声地聊着。

“你猜谁会胜出？”*ADO.NET*对旁边的*Hibernate*说。

“我觉得，抽象工厂可以解决多个类型产品的创建问题，就我而言，同一对象与多个数据库ORM就是通过她来实现的。我觉得她会赢。”*Hibernate*坚定地说。

“你也不看看，抽象工厂那形象，多臃肿呀，身上类这么多。做起事来一定不够利索。”*ADO.NET*不喜欢抽象工厂。

“那你喜欢单例？”*Hibernate*问道。

“单例又太瘦了。过于骨感也不是美。我其实蛮喜欢原型那小姑娘的，我的DataSet只要调用原型模式的Clone就可以解决数据结构的复制问题，而Copy则不但复制了结构，连数据也都复制完成，很是方便。”

“那你不觉得建造者把建造过程隐藏，一个请求，完整产品就创建，在高内聚的前提下使得与外界的耦合大大降低，这不也是很棒的吗？”

“问题是又有多少产品是相同的建造过程呢？再说回来，你造什么对象，不还是需要new吗？”

“哈，从new的角度讲，工厂方法才是最棒的设计，它可是把工厂职责都分了类了，其他几位不过是她的变体罢了。”

“有点道理，看来创建型这一组，工厂方法有点优势哦。”

“下面有请评委提问。”主持人GOF待五位选手出场亮相之后接着说。

“请问抽象工厂小姐，为什么我们需要创建型模式？”开放-封闭先生问道。

只见抽象工厂思考了一下，说道：“我觉得创建型模式隐藏了这些类的实例是如何被创建和放在一起，整个系统关于这些对象所知道的是由抽象类所定义的接口。这样，创建型模式在创建了什么、谁创建它、它是怎么被创建的，以及何时创建这些方面提供了很大的灵活性**[DP]**。”

“请问原型小姐，你有什么补充？”依赖倒转对着原型问道。

原型显然没想到突然会问到她，而且对于这个问题，多少有点手足无措，她说：“当一个系统应该独立于它的产品创建、构成和表示时，应该考虑用创建性模式。建立相应数目的原型并克隆它们通常比每次用合适的状态手工实例化该类更方便一些**[DP]**。”

“哈，这可能是我们需要原型的理由。”依赖倒转说道，然后转头问建造者，“请谈谈你对松耦合的理解。”

建造者对这个问题一定是有了准备，不慌不忙，说道：“这个问题首先要谈谈内聚性与耦合性，内聚性描述的是一个例程内部组成部分之间相互联系的紧密程度。而耦合性描述的是一个例程与其他例程之间联系的紧密程度。软件开发的目标应该是创建这样的例程：内部完整，也就是高内聚，而与其他例程之间的联系则是小巧、直接、可见、灵活的，这就是松耦合**[DPE]**。”

“那么你自己是如何去实践松耦合的呢？”依赖倒转接着问。

“我是将一个复杂对象的构建与它的表示分离，这就可以很容易地改变一个产品的内部表示，并且使得构造代码和表示代码分开。这样对于客户来说，它无需关心产品的创建过程，而只要告诉我需要什么，我就能用同样的构建过程创建不同的产品给客户 **[DP]** 。”

“回答得非常好，现在请问单例，你来说说看你参赛的理由，你与别人有何不同？”单一职责问道。

单例小姐有些羞涩，停了一会，才开口说：“我觉得对一些类来说，一个实例是很重要的。一个全局变量可以使得一个对象被访问，但它不能防止客户实例化多个对象。我的优势就是让类自身负责保存它的唯一实例。这个类可以保证没有其他实例可以被创建，并且我还提供了一个访问该实例的方法。这样就使得对唯一的实例可以严格地控制客户怎样以及何时访问它 **[DP]** 。”

“工厂方法，请问你如何理解创建型模式存在的意义？”合成聚合复用问道。

此时只听场下一声音叫道，“二姐加油”，原来简单工厂在观众席上喊叫呢。工厂方法对着观众席微笑了一下，然后非常有信心地答道，“创建型模式抽象了实例化的过程。它们帮助一个系统独立于如何创建、组合和表示它的那些对象。创建型模式都会将关于该系统使用哪些具体的类的信息封装起来。允许客户用结构和功能差别很大的‘产品’对象配置一个系统。配置可以是静态的，即在编译时指定，也可以是动态的，就是运行时再指定。 **[DP]** ”

“那么请问你与其他几位创建型模式相比有什么优势？”

“我觉得她们几位都可能设计出比我更加灵活的代码，但她们的实现也相对就更加复杂。通常设计应该是从我，也就是工厂方法开始，当设计者发现需要更大的灵活性时，设计便会向其他创建型模式演化。当设计者在设计标准之间进行权衡的时候，了解多个创建型模式可以给设计者更多的选择余地。 **[DP]** ”

几位评委都在不住地点头，显然，他们非常肯定工厂方法的回答。

“下面有请几位评委写上您们认为表现最好的模式小姐。”*GOF*说道。

“单一职责先生，您的答案是？”

只见单一职责翻转纸牌，上面写着“单例”。

“非常好，单例小姐已有一票。”

“开放封闭先生，您的选择是？”

“工厂方法。我觉得工厂方法能使得我们增加新的产品时，不需要去更改原有的产品体系和工厂类，只需扩展新的类就可以了。这对于一个模式是否优秀是非常重要的判断标准，我选择她。”开放封闭说道。

“OK，工厂方法小姐也有一票了。”

……

“工厂方法小姐再加一票。”

……

“工厂方法小姐一共有五票。成功晋级，恭喜你。”*GOF*宣布完，只见工厂方法抱住了旁边的抽象工厂泪流满面，喜极而泣。下面的简单工厂和一帮工厂方法的小Fans们欢呼雀跃。而其他模式的Fans们低头不语，个别竟然已潸然泪下。

“好的，各位来宾，观众朋友们，第一场的比赛现在结束，工厂方法成功晋级，但其他四位选手并不等于没有机会，希望您能通过手机给她们投票，移动用户，请发送OO加选手编号到www.ootv.com，联通用户请发送OO加选手编号到www.ootv.net，其他手机用户请发送OO加选手编号到www.ootv.org，您的支持将是对落选选手的最大鼓励，最终获得票数最多者同样可以晋级决赛。下面休息一会，插播一段广告。”

*ADO.NET*开始发牢骚：“什么最大鼓励，根本就是电视台在骗钱。”

“你不发拉倒，我可要给抽象工厂投上一票了。”*Hibernate*说道。

“嗨，算了，反正也就一元钱，我给原型投上一票。”

“哥们，原型没戏了，投抽象工厂吧，这样她进级了，你的钱也不白花。”

“呸，你怎么知道抽象工厂会比原型的票数多，大家都不投她，她能成功吗？我不但投原型，而且要投她十五张票（最高限额）。”*ADO.NET*坚持道。

“我碰到神经病了。你去打水漂去吧，我不陪你。”

29.5 结构型模式比赛

此时的后台，第二组选手正在做着准备，电视台的记者抓紧时间，对适配器小姐做了一个小小的专访。

“适配器小姐，您入选的结构型模式组被称为死亡之组，这一点您怎么看？”

“我觉得，所谓死亡之组，意思是有多个可能得冠军的选手不幸被分在了一组，造成有实力的选手会在小组赛中提前被淘汰，但那是针对体育比赛，我们这种选秀活动，选手只要充分表现了自己，就是成功，最终结果往往是多赢的局面。所以我不担心。”

“您觉得您有可能成为冠军吗？”

“不想当冠军的模式不是好模式。我来了，当然就是要努力争取第一。”

“您是否有与众不同的杀手锏来获得胜利？”

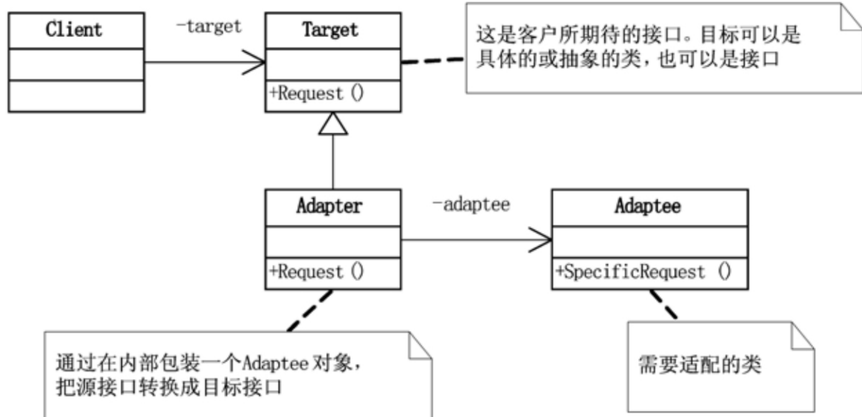
“我有杀手锏？”适配器小姐笑着摇摇头，“努力争取胜利就可以了。不好意思，我得准备去了，再见。”

当适配器离开后，记者小声地对摄像师说，“你把最后一句擦掉。然后我们再录。”接着这记者拿着话筒对着摄像机，正式地说道：“观众朋友，适配器小姐说，她有成功致胜的杀手锏，但她却并没有提及内容，最终是什么让我们拭目以待。OOTV记者赵谣前方为您报道。”

“欢迎回到第一届OOTV杯超级设计模式大赛现场，下面是第二组，也就是结构型模式组的比赛，她们将穿C#休闲装出场。”

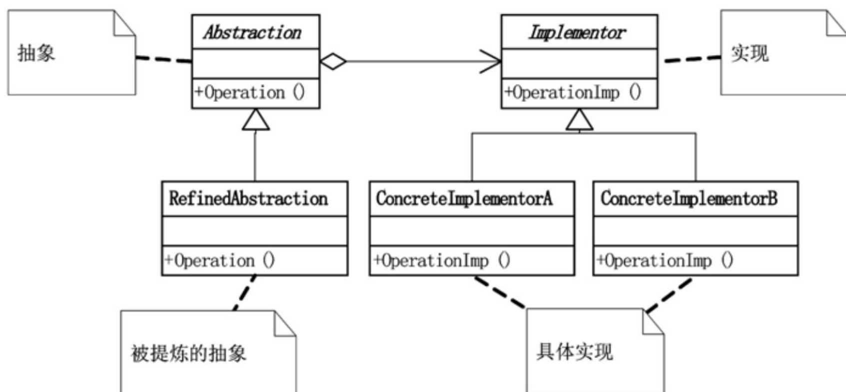
“6号选手，适配器小姐，她的口号是将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。 **[DP]** ”

6号选手 适配器（Adapter）



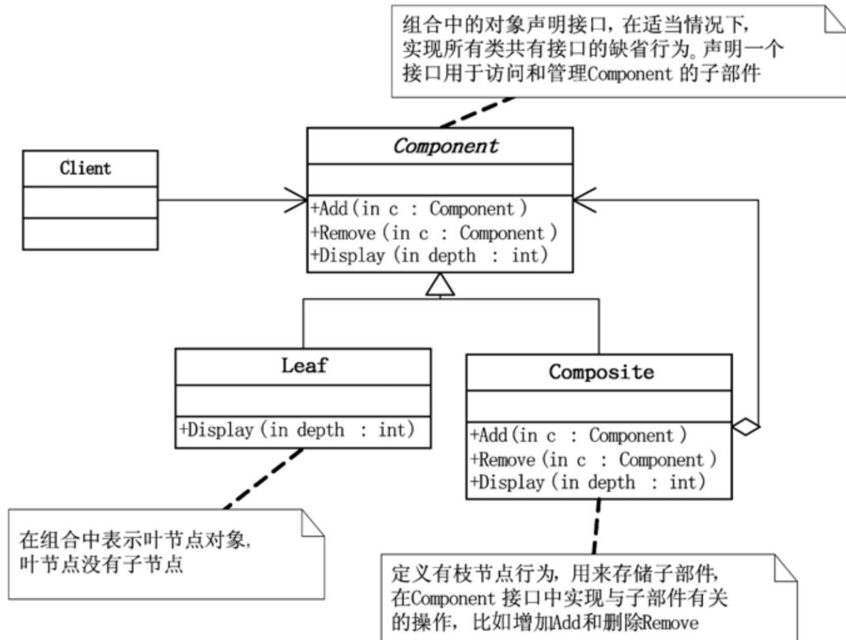
“7号选手叫桥接。桥接小姐提倡的是将抽象部分与它的实现部分分离，使它们都可以独立地变化。**[DP]**”

7号选手 桥接（**Bridge**）



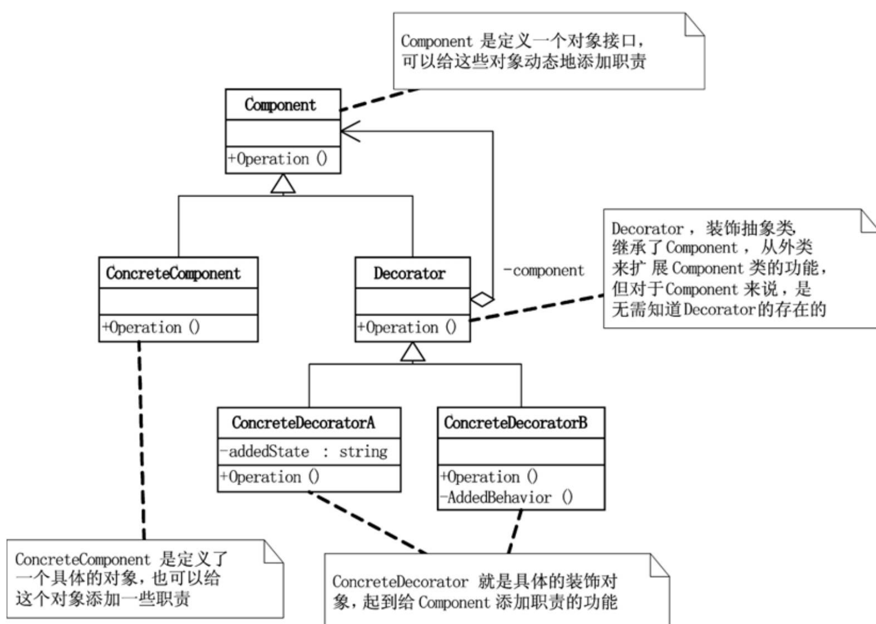
“8号选手向我们走来，组合小姐，一个非常美丽的姑娘，她的口号是将对象组合成树形结构以表示‘部分-整体’的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性。**[DP]**”

8号选手 组合（**Composite**）



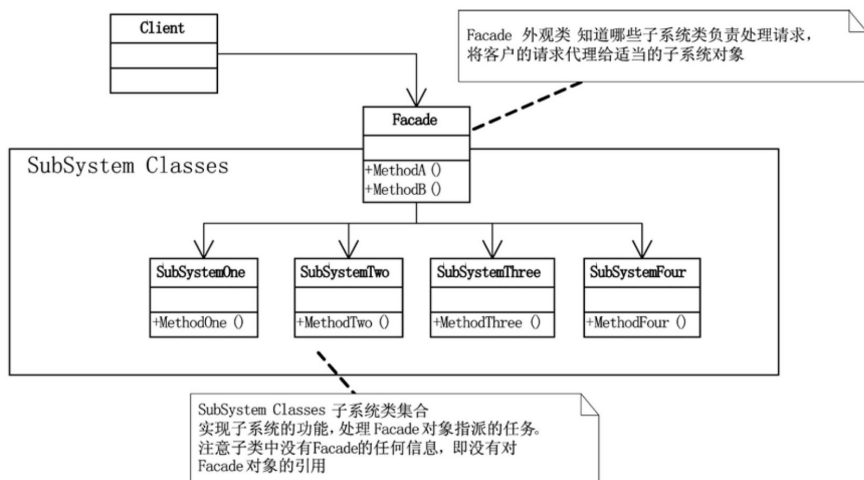
“9号选手，装饰小姐，她的意图非常简单，就是动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更加灵活 **[DP]** 。的确，她把自己装饰得非常漂亮。”

9号选手 装饰（**Decorator**）



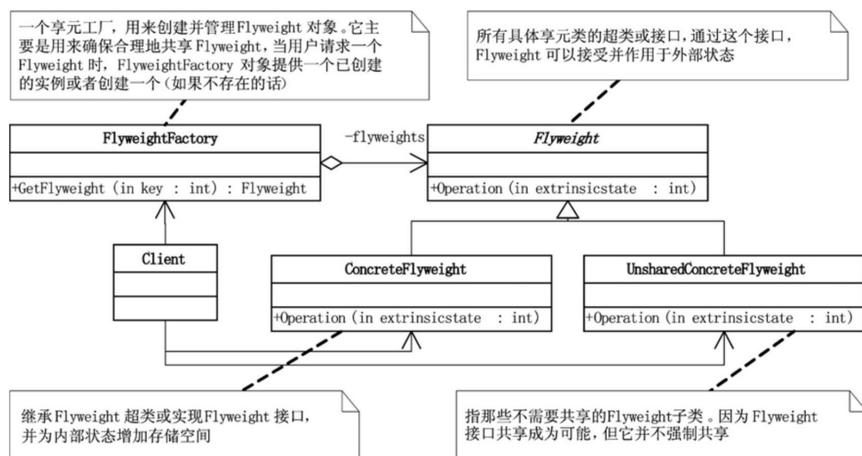
“10号选手出现了，外观小姐，她的形象如她的名字一样的棒，她说为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。**[DP]**”

10号选手 外观（Facade）



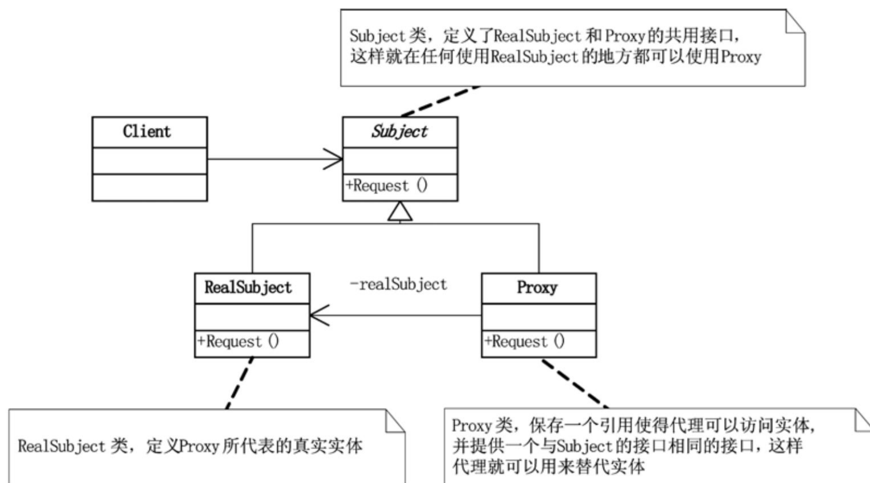
“11号选手是享元小姐，她的参赛宣言为运用共享技术有效地支持大量细粒度的对象。**[DP]**”

11号选手 享元（Flyweight）



“本组了最后一位，12号选手，代理小姐向我们走来，她声称为其他对象提供一种代理以控制对这个对象的访问。**[DP]**”

12号选手 代理（Proxy）



观众席中的*ADO.NET*和*Hibernate*又开始了讨论。

Hibernate：“C#休闲装我不喜欢，还是Java正装漂亮。”

ADO.NET：“哈，你这么正儿八经的人，当然是不喜欢休闲装，你兄弟*NHibernate*一定喜欢得不得了，我也是喜欢休闲装的。”

Hibernate：“这一组够强，没有太弱的，你感觉谁最有机会？”

ADO.NET：“不好说，都很漂亮的，各自有各自的特点，你认为呢？”

Hibernate：“我喜欢桥接，太漂亮了，那种解耦的方式，用聚合来代替继承，实在是非常巧妙。”

ADO.NET：“是的，桥接很漂亮，不过装饰也非常美丽。由于善于打扮，所以她可以很好地展示其魅力。”

Hibernate：“说得也是，装饰好歹也是靠化妆自己展示好看，而代理那个小妮子，听说她甚至有可能就不是自己来参加比赛，而是找了一替身。”

ADO.NET：“啊，不会吧，这种谣传你也会信呀，找人替身，那出了名算谁的？”

Hibernate：“当然还是她自己，大牌明星都这样的，找了替身做了很多，最后可能连替身名字都不让人家知道。”

ADO.NET：“代理要是大牌就不用来参加比赛了，出名前一切还是只有靠自己的。说心里话，我最喜欢的是适配器小姐。”

Hibernate："哦，为什么喜欢她？她好像并不算漂亮。"

ADO.NET："因为她对我的帮助最大，我在访问不同的数据库，如SQL Server、Oracle或者DB2等时，需要将数据结构和数据都转化成XML格式给DataSet，DataAdapter就是适配器，没有她的帮助，我的DataSet就发挥不了作用，真的很感激她。"

Hibernate："哈，原来是恩人呀，打小认识的？青梅竹马？哈，好像就你认识她一样，我也和她是老相好哦。"

ADO.NET："你就吹吧你，之前也听你说你和抽象工厂是相好，现在又和适配器相好，你的相好真够多的。"

Hibernate："不信拉倒。我们不如来打赌吧，我赌10块钱，桥接会赢。"

ADO.NET："瞧你那小气样，我赌100元，适配器会赢。"

Hibernate："100就100，Who怕Who呀。"

"下面有请评委提问。"主持人*GOF*说。

"请问适配器小姐，刚才记者提到你有成功的杀手锏，那是什么呢？"开放封闭先生问道。

"杀手锏？"适配器心里一咯噔，心想，"那记者太不道义了，我明明没有回答她的问题，怎么就断章取义地说我有杀手锏呢？造谣呀。"犹豫了一下，她说道，"我所谓的杀手锏是说，面向对象的精神就是更好地应对需求的变化，而现实中往往会有下面这些情况，想使用一个已经存在的类，而它的接口不符合要求，或者希望创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类协同工作。正如开放封闭先生您所倡导地对修改关闭，对扩展开放的原则，我可以做到让这些接口不同的类通过适配后，协同工作。**[DP]**"

开放封闭不住地点头。

"桥接小姐，面对变化，你是如何做的？"合成聚合复用问道。

桥接答道："继承是好的东西，但往往会过度地使用，继承会导致类的结构过于复杂，关系太多，难以维护，而更糟糕的是扩展性非常差。而仔细研究如果能发现继承体系中，有两个甚至多个方向的变化，那么就解耦这些不同方向的变化，通过对象组合的方式，把两个角色之间的继承关系改为了组合的关系，从而使这两者可以应对各自独立的变化，事实上也就是合成聚合复用女士所提倡的原则，总之，面对变化，我主张'找出变化并封装之'。**[DPE]**"

“这个问题也同样提问给装饰小姐，面对变化，你如何做？”合成聚合复用接着问装饰。

装饰显然对此问题很有信心，答道：“面对变化，如果采用生成子类的方法进行扩充，为支持每一种扩展的组合，会产生大量的子类，使得子类数目呈爆炸性增长。这也是刚才桥接小姐所提到的继承所带来的灾难，而事实上，这些子类多半只是为某个对象增加一些职责，此时通过装饰的方式，可以更加灵活、以动态、透明的方式给单个对象添加职责，并在不需要时，撤销相应的职责。[DP]”

“组合小姐，我们通过你的材料，了解到你最擅长于表示对象的部分与整体的层次结构。那么请问，你是如何做到这一点的？”里氏代换问道。

组合答道：“我是希望用户忽略组合对象与单个对象的不同，用户将可以统一地使用组合结构中的所有对象。”组合回答道，“用户使用组合类接口与组合结构中的对象进行交互，如果接收者是一个叶节点，则直接处理请求，如果接收者是组合对象，通常将请求发送给它的子部件，并在转发请求之前或之后可能执行一些辅助操作。组合模式的效果是客户可以一致地使用组合结构和单个对象。任何用到基本对象的地方都可以使用组合对象。**[DP]**”

一直没有提过问题的迪米特先生，突然接过话筒，对着外观小姐问了个问题，“请问外观小姐，信息的隐藏促进了软件的复用 **[J&DP]**，你怎么理解这句话？”

外观小姐有些紧张，停顿了一会，然后缓缓答道，“类之间的耦合越弱，越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成波及。如果两个类不必彼此直接通信，那么就不要让这两个类发生直接的相互作用。如果实在需要调用，可以通过第三者来转发调用。[J&DP]”

“那你又是如何去贯彻这一原则呢？”迪米特继续问道。

“我觉得应该让一个软件中的子系统间的通信和相互依赖关系达到最小，而具体办法就是引入一个外观对象，它为子系统间提供了一个单一而简单的屏障 **[DP]**。通常企业软件的三层或N层架构，层与层之间地分离其实就是外观模式的体现。”外观小姐说话很慢，但显然准备过，并没说错什么。迪米特满意地点了点头。

“享元小姐，请问你如何看待很多对象使得内存开销过大的问题？”单一职责问道。

“对象使得内存占用过多，而且如果都是大量重复的对象，那就是资源的极大浪费 **[DP]**，会使得机器性能减慢，这个显然是不行的。”享元说，“面向对象技术有时会因简单化的设计而代价极大。比如文档处理软

件，当中的字符都可以是对象，而如果让文档中的每一个字符都是一个字符对象的话，这就会产生难以接受的运行开销，显然这是不合理也是没必要的。由于文档字符就是那么些字母、数字或符号，完全可以让所有相同的字符都共享同一个对象，比如所有用到‘a’的字符的地方都使用一个共享的‘a’对象，这就可以节约大量的内存。”

“OK，最后一位，代理小姐，请对比一下你和外观小姐，有哪些不同？与适配器小姐又区别在何处？”迪米特问道。

代理没有想到会问这样一个问题，而旁边就站着外观和适配器，如果说得不好，显然就是很得罪人的事，她思考了片刻，说道：“代理与外观的主要区别在于，代理对象代表一个单一对象而外观对象代表一个子系统；代理的客户对象无法直接访问目标对象，由代理提供对单独的目标对象的访问控制，而外观的客户对象可以直接访问子系统中的各个对象，但通常由外观对象提供对子系统各元件功能的简化的共同层次的调用接口。 **[R2P]** ”代理停了一下，然后接着说，“至于我与适配器，其实都是属于一种衔接性质的功能。代理是一种原来对象的代表，其他需要与这个对象打交道的操作都是和这个代表交涉。而适配器则不需要虚构出一个代表者，只需要为应付特定使用目的，将原来的类进行一些组合。 **[DP]** ”

“下面有请六位评委写上您们认为表现最好的模式小姐。”*GOF*说道。

桥接

适配器

外观

适配器

桥接

外观

“哦，各位来宾，观众朋友们，第二场结构型模式的比赛真是相当精彩，各位选手也都实力相当，难分伯仲，现在出现了‘桥接’、‘适配器’、‘外观’的比分均为两分的相同情况。根据比赛规则，她们三位需要站上PK台，进行PK。三位有请。”

“下面请三位各自说一说你比其他两位优秀的地方。适配器小姐先来。”

适配器说：“我主要是为了解决两个已有接口之间不匹配的问题，我不需要考虑这些接口是怎样实现的，也不考虑它们各自可能会如何演

化。我的这种方式不需要对两个独立设计的类中任一个进行重新设计，就能够使它们协同工作。**[DP]**”

“非常好，下面有请桥接小姐。”

“我觉得我和适配器小姐具有一些共同的特征，就是给另一对象提供一定程度的间接性，这样可以有利于系统的灵活性。但正所谓未雨绸缪，我们不能等到问题发生了，再去考虑解决问题，而是更应该在设计之初就想好应该如何做来避免问题的发生，我通常是在设计之初，就对抽象接口与它的实现部分进行桥接，让抽象与实现两者可以独立演化。显然，我的优势更明显。[DP]”

“OK，说得很棒，外观小姐，您有什么观点？”

“首先我刚听完两位小姐的发言，我个人觉得她们各自有各自的优点，并不能说设计之初就一定比设计之后的弥补要好，事实上，在现实中，早已设计好的两个类，过后需要它们统一接口，整合为一的事例也比比皆是。因此桥接和适配器是被用于软件生命周期的不同阶段，针对的是不同的问题，谈不上孰优孰劣。然后，对于我来说，和适配器还有些近似，都是对现存系统的封装，有人说我其实就是另外一组对象的适配器，这种说法是不准确的，因为外观定义的是一个新的接口，而适配器则是复用一个原有的接口，适配器是使两个已有的接口协同工作，而外观则是为现存系统提供一个更为方便的访问接口。如果硬要说我是适配，那么适配器是用来适配对象的，而我则是用来适配整个子系统的。也就是说，我所针对的对象的粒度更大。[DP]”

“各个观众朋友们，在评委宣布结果之前，希望您能通过浏览器给她们投票，IE用户，Firefox用户，……，您的支持将是对当前选手的最大鼓励，最终获得票数最多者同样可以进级决赛。现在插播一段广告。”主持人*GOF*说道。

此时场下观众席中的两位，*ADO.NET*和*Hibernate*已经争论得不可开交。

Hibernate：“哈，你我看好的人都在PK台上，不过我相信桥接一定会赢。”

ADO.NET：“那可不一定，没出结果之前，别乱下结论，桥接老说适配器不如她，你没见评委在摇头吗？”

Hibernate：“我坚信桥接一定会赢，你要是不服，我加赌50，也就是150。”

ADO.NET：“哟哟哟，就加50，神气什么呀，要赌就赌大一些，1000，我赌适配器进级。”

Hibernate：“1000太多了点了，500吧。”

ADO.NET：“我赌1000，我赢了，你给我1000，我输了，我给你500。”

Hibernate：“小子，你也太狠了。赌，1000就1000。——评委呀，你们一定要看清楚呀，桥接才是真正的美女呀。”

ADO.NET：“哼，走着瞧吧。”

回到现场，“下面有请单一职责先生宣布评委的决定。”*GOF*大声说道。

“根据我们六位评委的讨论，做出了艰难的决策，最终统一了思想，一致决定，”单一职责停了停，“外观小姐晋级。”

外观小姐眼含泪光，但却保持着镇定，显然胜利的喜悦并没有让她失去常态。

适配器和桥接都非常失望，想哭，却又不得不强忍住泪水，强颜欢笑，对外观表示祝贺。

场下的两位看不出什么失望，反而都有些高兴。

ADO.NET：“我没赢，也没让你赢，这次算是打平。”

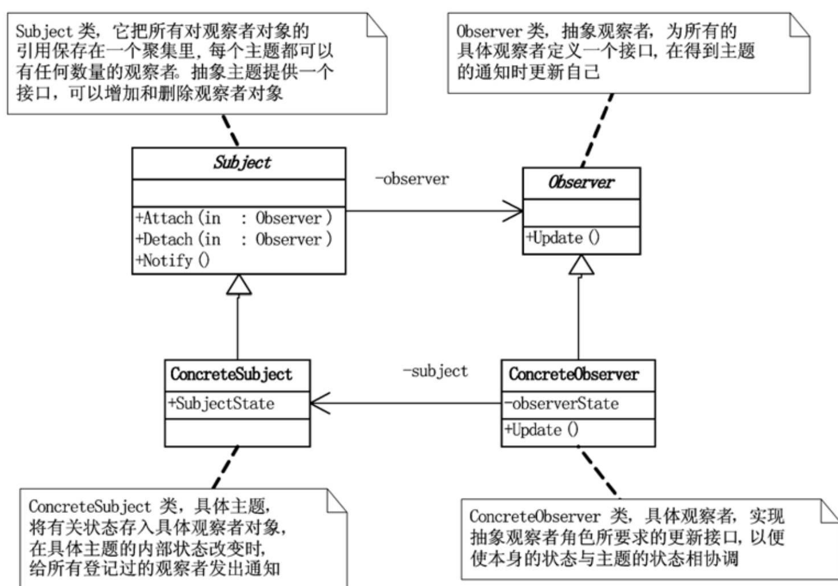
Hibernate：“没想到让这个小黑马冲了出来，真是奇怪呀。”

29.6 行为型模式一组比赛

“欢迎回到第一届OOTV杯超级设计模式大赛现场，下面是第三组，也就是行为型模式一组的比赛，她们将穿VB.NET运动装出场。”

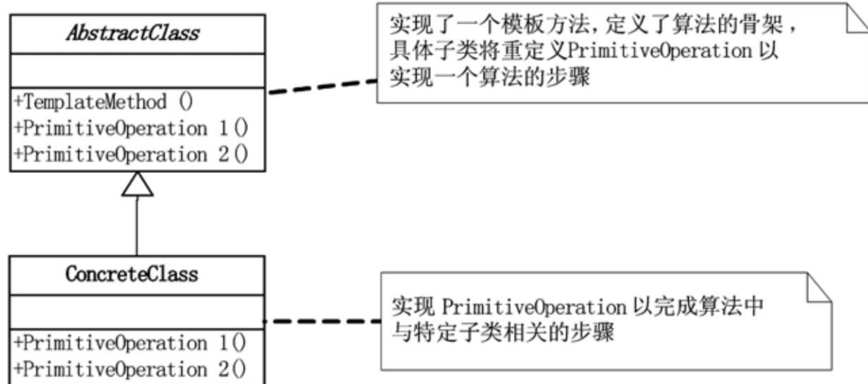
“首先出场的是13号选手，观察者小姐入场，它的口号是定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。**[DP]**”

13号选手 观察者（**Observer**）



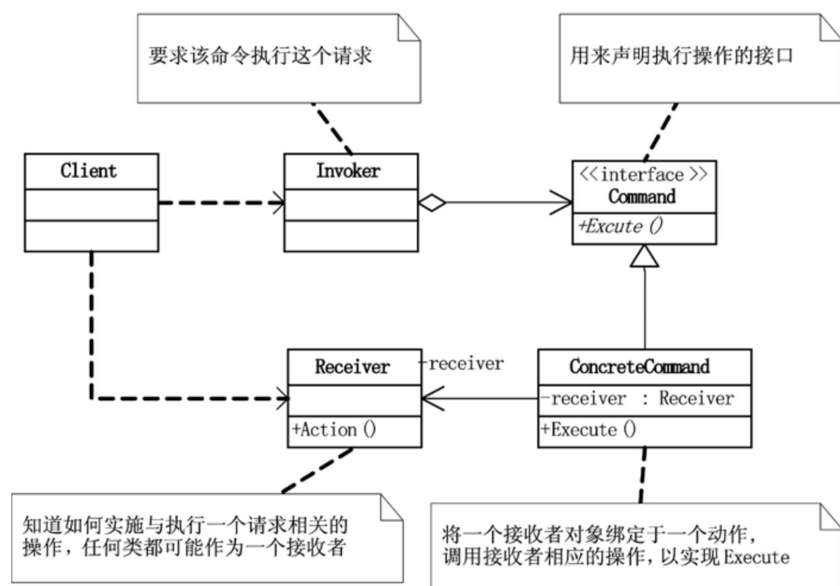
“14号选手，模板方法小姐，她提倡定义一个操作的算法骨架，而将一些步骤延迟到子类中，模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。**[DP]**”

14号选手 模板方法（**TemplateMethod**）



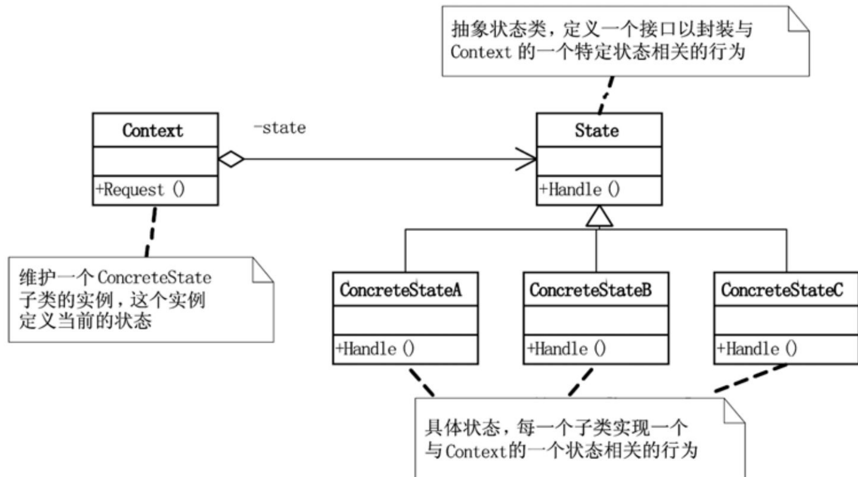
“15号选手是命令小姐，它觉得应该将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；可以对请求排队或记录请求日志，以及支持可撤销的操作。**[DP]**”

15号选手 命令（Command）



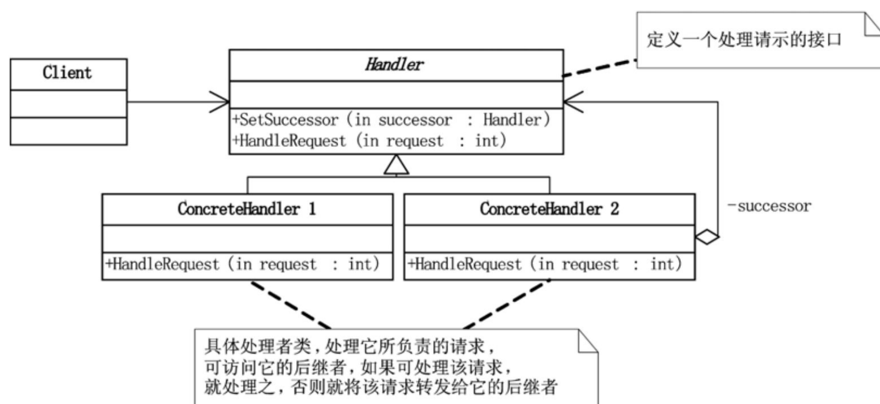
“16号是状态小姐，她说允许一个对象在其内部状态改变时改变它的行为，让对象看起来似乎修改了它的类。**[DP]**”

16号选手 状态（State）



“本组最后一位，17号选手，职责链小姐，她一直认为使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。 **[DP]** ”

17号选手 职责链（Chain of Responsibility）



观众席中的*ADO.NET*和*Hibernate*又开始了讨论。

Hibernate：“VB.NET是你们.NET家族的品牌吧？”

ADO.NET：“是的，最早是BASIC，它是很古老的一个品牌，经过二十多年的发展，它已经成功地从以简单入门为标准转到了完全面向对象，真的很不容易，即简单易懂，又功能强大，所以它做出的运动装非常实用。”

Hibernate：“行为型模式的小姐们长得怎么都不太一样，风格差异

也太大了。我不喜欢。”

ADO.NET：“我却觉得还不错，它们大多各有各的特长，比如这一组，应该还是有点看头，观察者、模板方法、命令都算是比较强的选手。”

Hibernate：“多半是观察者胜出了，因为她实在是到处接拍广告，做宣传，什么地方都能见到她的踪影，恨不得通知所有人，她是一设计模式。”

ADO.NET：“我猜也是她，人家本来就是以通知为主要魅力点的模式呀。咱们拭目以待吧。”

“下面有请评委提问。”主持人*GOF*说。

“请问观察者小姐，说说你对解除对象间的紧耦合关系的理解？”依赖倒转问道。

“我觉得对象间，尤其是具体对象间，相互知道的越少越好，这样发生改变时才不至于互相影响。对于我来说，目标和观察者不是紧密耦合的，它们可以属于一个系统中的不同抽象层次，目标所知道的仅仅是它有一系列的观察者，每个观察者实现Observer的简单接口，观察者属于哪一个具体类，目标是不知道的。”

“非常好，请问模板方法小姐，请你谈谈，你对代码重复的理解以及你如何实现代码重用？”里氏代换问。

模板方法说，“代码重复是编程中最常见、最糟糕的‘坏味道’，如果我们在一个以上的地方看到相同的程序结构，那么可以肯定，设法将它们合而为一，程序会变得更好[RIDEC]。但是完全相同的代码当然存在明显的重复，而微妙的重复会出现在表面不同但是本质相同的结构或处理步骤中 **[R2P]** ，这使得我们一定要小心处理。继承的一个非常大的好处就是你能免费地从基类获取一些东西，当你继承一个类时，派生类马上就可以获得基类中所有的功能，你还可以在它的基础上任意增加新的功能。模板方法模式由一个抽象类组成，这个抽象类定义了需要覆盖的可能有不同实现的模板方法，每个从这个抽象类派生的具体类将为此模板实现新方法 **[DPE]** 。这样就使得，所有可重复的代码都提炼到抽象类中了，这就实现了代码的重用。”

“下面请问命令小姐，为什么要将请求发送者与具体实现者分离？这有什么好处？”单一职责问道。

“您的意思其实就是将调用操作的对象与知道如何实现该操作的对象解耦，而这就意味着我可以在这两者之间处理很多事，比如完全可以发送者发送完请求就完事了，具体怎么做是我的事，我可以在不同的时刻指定、排列和执行请求。再比如我可以在实施操作前将状态存储起来，

以便支持取消/重做的操作。我还可以记录整个操作的日志，以便以后可以在系统出问题时查找原因或恢复重做。当然，这也就意味着我可以支持事务，要么所有的命令全部执行成功，要么恢复到什么也没执行的状态。总之，如果有类似的需求时，利用命令模式分离请求者与实现者，是最明智的选择。”

“OK，职责链小姐，提问命令小姐的问题同样提问给你，为什么要将请求发送者与具体实现者分离？这有什么好处？你如何回答。”

“我们时常会碰到这种情况，就是有多个对象可以处理一个请求，哪个对象处理该请求事先并不知道，要在运行时刻自动确定，此时，最好的办法就是让请求发送者与具体处理者分离，让客户在不明确指定接收者的情况下，提交一个请求，然后由所有能处理这请求的对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。”职责链答道，“比如我住在县城，生了怪病，我不知道什么级别的医院可以诊治，显然最简单的办法就是马上找附近的医院，让此医院来决定是否可以治疗，如果不能则医院会提供转院的建议，由县级转市级、由市级转省级、由省级转国家级，反正直到可以治疗为至。这就不需要请求发送者了解所有处理者才能处理问题了。”

“非常好，例子很形象，不过得怪病不是好事，健康才最重要。”开放封闭微笑道，“下面请问最后一位，状态小姐，条件分支的大量应用有何问题？如何正确看待它？”

状态答道：“如果条件分支语句没有涉及重要的商务逻辑或者不会随着时间的变化而变化，也不会有任何的可扩展性，换句话说，它几乎不会变化，此时条件分支是应该使用的。但是注意我这里用到了很多前提，这些前提往往都是不成立的，事实上不会变化的需求很少，不需要扩展的软件也很少，那么如果把这样的分支语句进行分解并封装成多个子类，利用多态来提高其可维护、可扩展的需要，是非常重要的。状态模式提供了一个更好的办法来组织与特定状态相关的代码，决定状态转移的逻辑不在单块的**if**或**switch**中，而是分布在各个状态子类之间，由于所有与状态相关的代码都存在于某个状态子类中，所以通过定义新的子类可以很容易地增加新的状态和转换。**[DP]**”

“下面有请六位评委写上你们认为表现最好的模式小姐。”*GOF*说道。

“喔，观察者3票、模板方法2票、命令1票，最终观察者小姐晋级。”*GOF*在等评委翻牌后宣布道。“恭喜你，观察者小姐，有什么要说的吗？”

观察者小姐平静地说：“感谢所有关心我、喜欢我和憎恨我的人。比赛中的环境不太干净，但我是干净地站起来的。”

“啊，你，你说憎恨？不干净？什么意思？”*GOF*非常意外。

“无可奉告。”观察者显然知道刚才那些话的影响，所以选择回避。

“哦，”*GOF*有些尴尬，“下面先进段广告，广告后我们进行第四组的比赛。”慌忙之中，*GOF*连让短信投票的宣传都忘记说了。

此时场下观众都议论纷纷。当然*ADO.NET*和*Hibernate*两位也不例外。

ADO.NET：“你说她被谁憎恨了？”

Hibernate：“那谁知道。不过一定是来参赛前，受到了一些阻挠，甚至于产生了很大的矛盾，因此才有了憎恨一说。其实憎恨也就罢了，‘不干净’一词力道可就重了。”

ADO.NET：“这有什么，只不过观察者她胆子大，说了出来。在娱乐圈、体育圈有潜规则，难道我们程序世界里就没有潜规则？”

Hibernate：“是呀，只要涉及到利益，就不可能没有交易。我再给你爆个料，*MVC*你听说过吗？”

ADO.NET：“知道呀，大名鼎鼎的*MVC*模式，就是Model/View/Controller，非常漂亮的姑娘。在电视上经常能看到它，好像谈模式、谈架构没有不谈到她的。”

Hibernate：“你可知道为何她没来参加这次超级模式大赛？”

ADO.NET：“咦，对哦，为什么她没来参加呢，要是她来，和这23个比，至少前三是一定可以进的。你不要告诉我，因为她被潜规则了？”

Hibernate：“我偷偷告诉你，你可别出去乱传。**MVC**是包括三类对象，**Model**是应用对象，**View**是它在屏幕上的表示，**Controller**定义用户界面对用户输入的响应方式。如果不使用**MVC**，则用户界面设计往往将这些对象混在一起，而**MVC**则将它们分离以提高灵活性和复用性 **[DP]**。因此，有人甚至说，她是集观察者、组合、策略三个美女优点于一身的靓女。海洗和选拔赛时她都表现非常好的，但因为一次短信的事情，而她又在自己博客里写了《非得这样吗？》的文章，大大地得罪了主办方的一个大鳄。于是由于这件事，她就彻底把自己的前途给葬送了。后来博客的文章也被勒令删除。”

ADO.NET：“得罪谁了？短信什么内容？”

Hibernate：“我哪知道呀，反正她后来就退出比赛了。”

ADO.NET：“你这也叫爆料呀，什么都没说出来，根本就一个听风是雨没任何根据的小道消息。据我猜测，主要原因是这次是设计模式比赛，而**MVC**是多种模式的综合应用，应该算是一种架构模式，所以被排

除在外。”

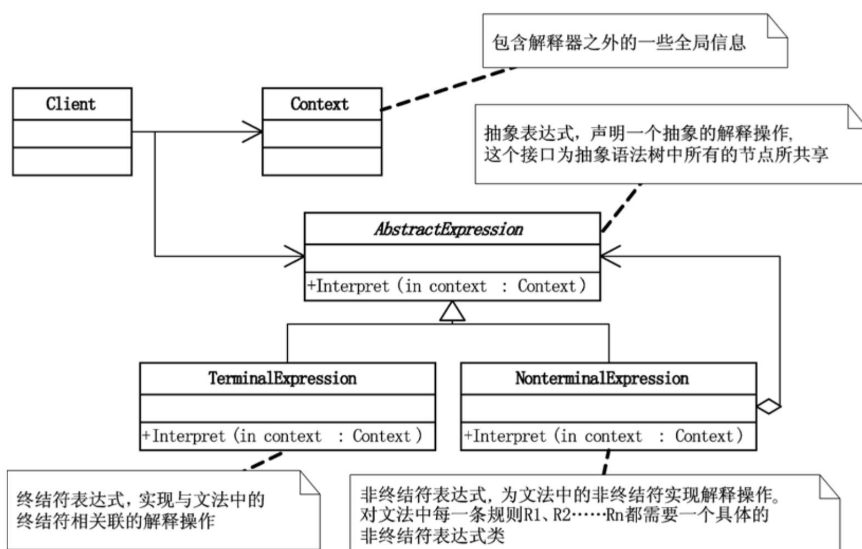
Hibernate：“不信就算了，不过你说的也有道理。”

29.7 行为型模式二组比赛

“欢迎回到第一届OOTV杯超级设计模式大赛现场，下面是行为型模式二组，也就是最后一组的比赛，她们将穿C＋＋旗袍出场。”

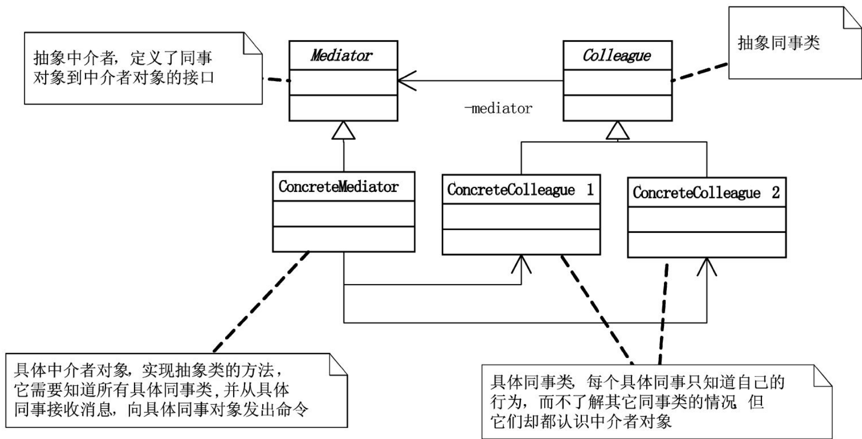
“首先出场的是18号选手，解释器小姐，它声称给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。**[DP]**”

18号选手 解释器（**interpreter**）



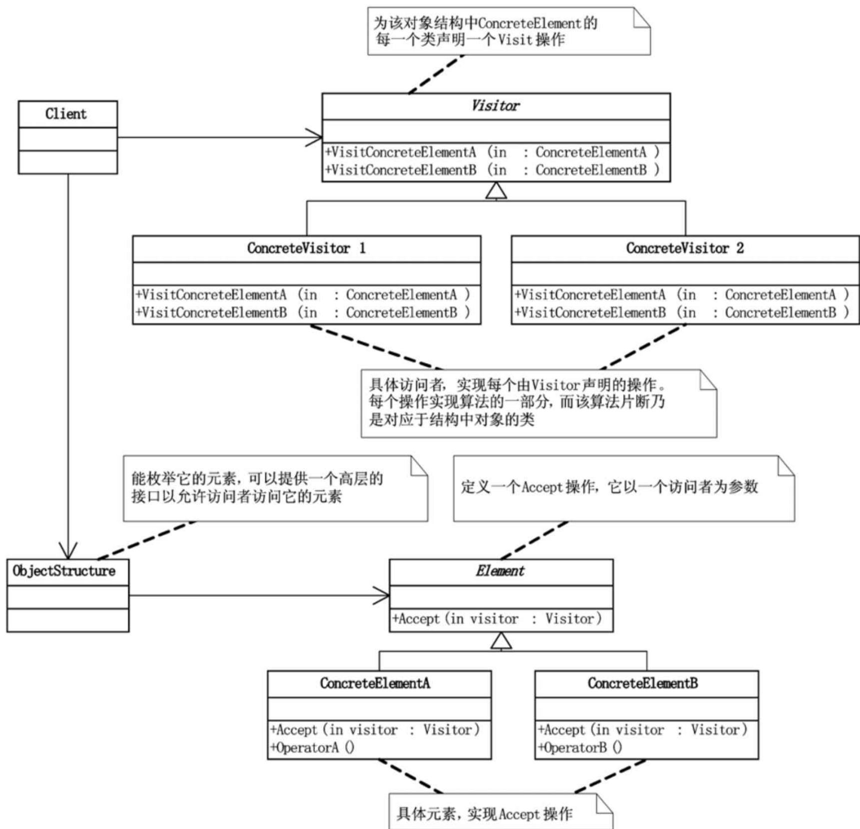
“19号选手是中介者小姐，她说她是用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。**[DP]**”

19号选手 中介者（**Mediator**）



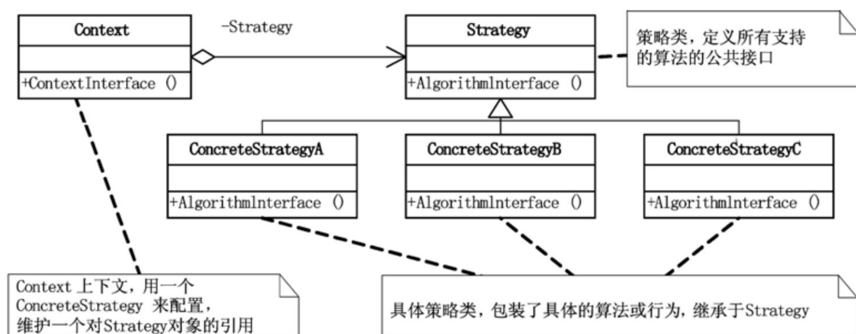
“20号小姐向我们走来，访问者小姐，她表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。**[DP]**”

20号选手 访问者（Visitor）



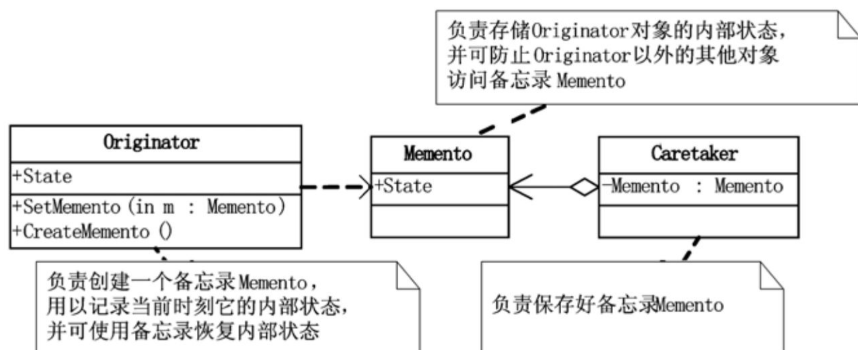
“21号小姐是策略，一个可爱的姑娘，她的意图是定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。 **[DP]** ”

21号选手 策略（**Strategy**）



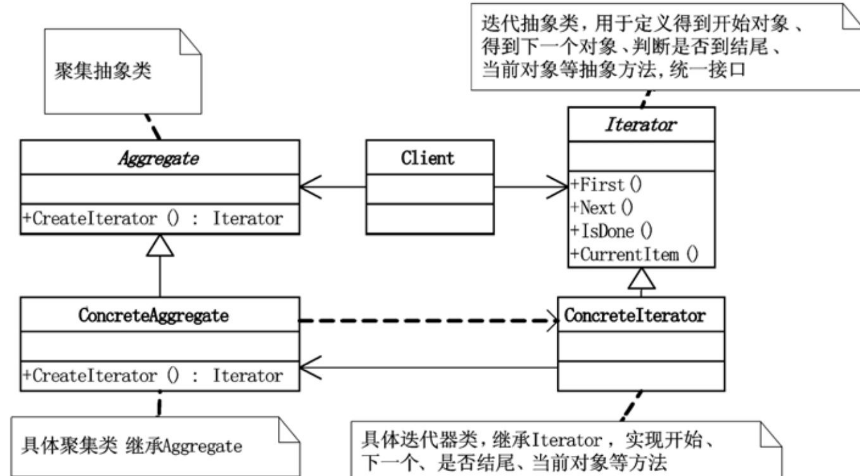
“22号选手，备忘录小姐，她说在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。 **[DP]** ”

22号选手 备忘录（**Memento**）



“最后一名选手，23号，迭代器小姐，她说，提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。 **[DP]** ”

23号选手 迭代器（**Iterator**）



Hibernate："这组里我只认识策略小姐，看过她做过不少广告，迭代器好像也听说过。其他的MM太没名气了，我不看好她们。"

ADO.NET："中介者也还算行吧，至少我是知道她的。不过这一组实力是不太强，估计策略拿第一没什么悬念了。"

"好的，各位小姐已展示完毕，下面有请评委提问。"主持人*GOF*说。

"请问解释器小姐，说说你参赛的动机和优势？"依赖倒转问道。

解释器小姐很镇定地答道："在编程世界里，实现目标都是通过编写语言并执行来实现的，从最低级的机器语言到人能很容易读懂机器也可以执行的高级语言，但是高级语言编写起一些问题可能还是比较复杂。如果一种特定类型的问题发生的频率足够高，那么就可以考虑将该问题的各个实例表述为一个简单语言中的句子。也就是说，通过构建一个解释器，该解释器解释这些句子来解决该问题 **[DP]** 。比如正则表达式就是描述字符串模式的一种标准语言，与其为每一个字符串模式都构造一个特定的算法，不如使用一种通用的搜索算法来解释执行一个正则表达式，该正则表达式定义了待匹配字符器的集合[DP]。"

"中介者小姐，人家都说你是交际花，请问你广交朋友的目的是什么？"迪米特问道。

"交际花不敢当，但我的确喜欢交朋友。面向对象设计鼓励将行为分布到各个对象中，这种分布可能会导致对象间有许多连接。也就是说，有可能每一个对象都需要知道其他许多对象。对象间的大量相互连接使得一个对象似乎不太可能在没有其他对象的支持下工作，这对于应对变化是不利的，任何较大的改动都很困难 **[DP]** 。所以说朋友多既是好事情，其实也是坏事情。我提倡将集体行为封装一个单独的中介者对象来

避免这个问题，中介者负责控制和协调一组对象间的交互。中介者充当一个中介以使组中的对象不再相互显式引用。这些对象仅知道中介者，从而减少了相互连接的数目 **[DP]** 。我作为中介者，广交朋友，就起到了在朋友间牵线搭桥的作用。可以为各位朋友们服务。这其实不也正是迪米特先生您一直倡导的最少知识原则，也就是如何减少耦合的问题，类之间的耦合越弱，越有利于复用 **[J&DP]** 。”

现在轮到访问者了，她面无表情，不知是否很紧张。合成聚合复用问道：“访问者小姐，听说你对朋友要求很苛刻，要请到你帮忙是很难的事情，你喜欢交朋友吗？”

听到这个问题，访问者笑开了颜：“哪有这种事情，朋友要我帮忙，我都会尽力而为的。的确，我不太喜欢交很多朋友，一般找到好朋友了，就不喜欢再交往新的朋友了。我的理念是朋友在精不在多。但是我和朋友间的交往通常会是多方面的，一同聊天、逛街、旅游、唱歌、游泳，哪怕是我们不会的活动，我们也可以尝试一起去学习、去扩展我们的生活情趣。也就是说，访问者增加具体的**Element**是困难的，但增加依赖于复杂对象结构的构件的操作就变得容易。仅需增加一个新的访问者即可在一个对象结构上定义一个新的操作。”

“非常有意思的交友观。下面请策略小姐准备接受提问。”*GOF*说道。

“请问策略小姐，说说你对‘优先使用对象组合，而非类继承’的理解？”合成聚合复用问道。

策略小姐答得很流利，“继承提供了一种支持多种算法或行为的方法，我们可以直接生成一个类**A**的子类**B**、**C**、**D**，从而给它以不同的行为。但这样会将行为硬行编制到父类**A**当中，而将算法的实现与类**A**的实现混合起来，从而使得类**A**难以理解、难以维护和难以扩展，而且还不能动态地改变算法。仔细分析会发现，它们之间的唯一差别是它们所使用的算法或行为，将算法封装在独立的策略**Strategy**类中使得你可以独立于其类**A**改变它，使它易于切换、易于理解、易于扩展 **[DP]** 。这里显然使用对象组合要优于类继承。”

“策略小姐说得非常好，下面想请问一下备忘录小姐，在保存对象的内部状态时，为何需要考虑不破坏封装细节的前提？”单一职责问道。

“通常原对象A都有很多状态属性，保存对象的内部状态，其实也就是将这些状态属性的值可以记录到A对象外部的另一个对象B，但是，如果记录的过程是对外透明的，那就意味着保存过程耦合了对象状态细节。使用备忘录就不会出现这个问题，它可以避免暴露一些只应由对象**A**管理却又必须存储在对象**A**之外的信息。备忘录模式把可能很复杂的对象**A**的内部信息对其他对象屏蔽起来，从而保持了封装边界 **[DP]** 。”

“最后一位，迭代器小姐，说说迭代器模式对遍历对象的意义？”里

氏代换问道。

“一个集合对象，它当中具体是些什么对象元素我并不知道，但不管如何，应该提供一种方法来让别人可以访问它的元素，而且可能要以不同的方式遍历这个集合。迭代器模式的关键思想是将对列表的访问和遍历从列表对象中分离出来并放入一个迭代器对象中，迭代器类定义了一个访问该列表元素的接口。迭代器对象负责跟踪当前的元素，并且知道哪些元素已经遍历过了**[DP]**。”

“下面有请六位评委评选出行为型二组比赛的第一名。”*GOF*说道。

“迭代器小姐2票、策略小姐4票。”*GOF*宣布。“晋级的是策略小姐。”

“Yeah!”策略小姐右手伸出食指和中指，打了“V”型手势，向台前晃了晃，然后收到头顶上方握紧拳头做下拉状。

“哈，策略小姐高兴起来真像个孩子，说说感受吧。”*GOF*对策略的反应也很开心，微笑着说。

“我要感谢OOTV，感谢六位评委，感谢咱爸咱妈，感谢所有支持我的朋友，我爱你们!”策略仿佛已经问鼎冠军一样说了一大堆感谢的话。

“各位观众，请加快给你喜爱的选手投票，广告过后，我们将关闭短信通道。宣布投票结果。”*GOF*宣布说。

听主持人一宣布，下面的Fans们又纷纷掏出手机开始最后一轮的疯狂短信。

Hibernate：“看来我们英雄所见完全相同。”

ADO.NET：“是呀，策略早就成名了，所以很习惯于这种大场合说些场面话，明星也是练出来的。”

Hibernate：“现在工厂方法、外观、观察者、策略晋级了，除了这几位，你猜短信的结果是谁？”

ADO.NET：“从感觉上来讲，组合、命令、适配器、迭代器都有机会。”

Hibernate：“就不会出黑马？比如抽象工厂、代理、模板方法、桥接等？不过这次确实难料了。”

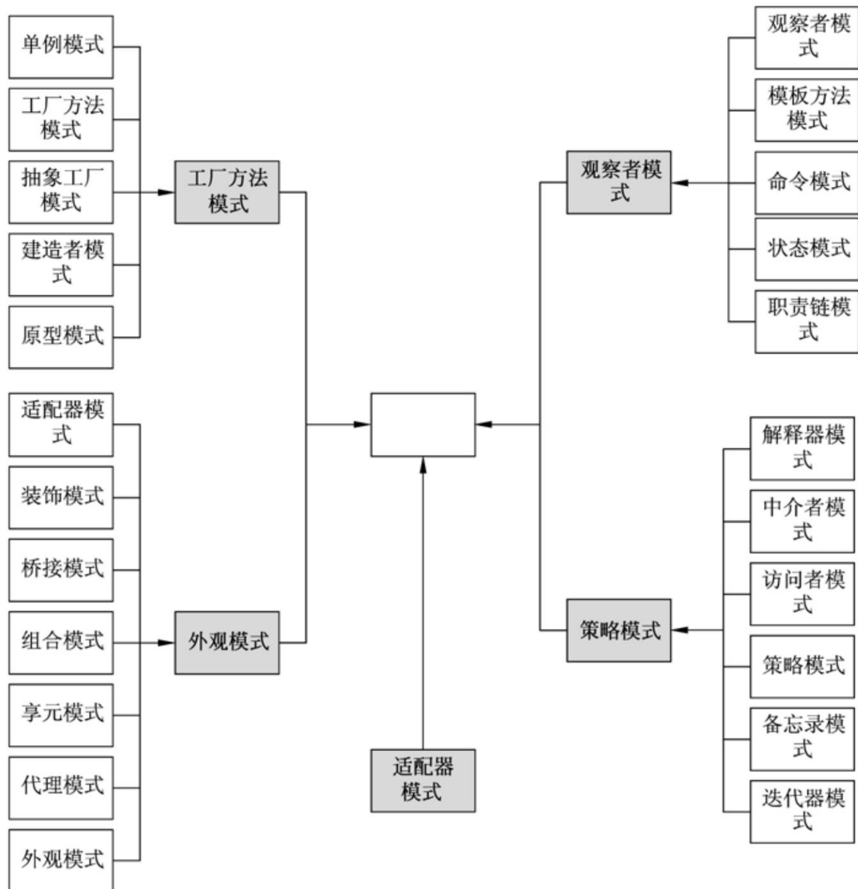
29.8　决赛

“欢迎回来，我们的短信平台刚刚关闭，结果已经出来。”*GOF*说，“除了四位入选的选手外，短信票数最多的选手是……她是……适配器小姐。”

适配器手捧住脸，刚才PK失利时都没有流泪的她，现在却梨花带雨，楚楚动人。相信她自己也没想到竟然会是自己，所有的人都将目光集中到了这个被称为有“杀手锏”的女孩身上。

“谢谢，谢谢观众朋友，我真心地……感谢……您们。”适配器有些激动，有些泣不成声，和刚才的策略小姐的表现形成了鲜明的对比。

“好的，现在我们比赛已经进入了最高潮，23位选手，经过激烈的比拼，现在选出了五位选手将站在PK台上，来决定今天冠亚季军的归属。她们分别是……工厂方法小姐、外观小姐、观察者小姐、策略小姐和观众朋友选出的可爱的适配器小姐。”



“决赛是一道应用题，希望五位选手根据你们的经验，来说明如果你参与其中，将会发挥什么样的作用，具体如何做。评委将根据各位的临场表现打分，最终评出冠亚季军。”*GOF*说道，“各位请听题。”

“题目是说有一家以软件开发和服务为主的创业型的公司创业初期初见成效，公司有了发展，员工数量大量增加，于是就考虑自主开发一套薪资管理系统来对公司的员工薪资进行管理，做得好也将作为产品对外销售。公司的员工按照薪资来分类大致可分为普通员工，他们按照月薪制发放，每月有工资和奖金；市场和销售人员，按照每月底薪加提成的方式发放；中高级管理人员，按照年薪加分红的方式发放；兼职人员和临时工，按照小时计费发放。系统要求界面不花哨，但要方便易用，如要有菜单、工具栏和状态栏等要素。产品初步打算用SQL Server作为数据库，但不排除未来成为产品后将可以应用于Oracle、MySQL等数据库来做数据持久化。薪资可以提供多种查询和统计的功能，报表能生成各种复杂的统计图。由于系统是自主研发的产品，所以统计图生成如若时间充裕则自行开发，否则就购买第三方的组件。”*GOF*念完题目后，接着说，“好了，现在请选手们准备一下，说出你们对这个系统设计的建议。”

“首先有请外观小姐发言。”

“我觉得这是一套典型的企业办公软件，所以在设计上我建议用三层架构比较好，也就是表示层、业务逻辑层和数据访问层。由于公司对业务有明确的需求，但对界面却有模糊性，‘不花哨’和‘方便易用’实在是仁者见仁，智者见智。所以我的建议是在业务逻辑与表示层之间，增加一业务外观层，这样就让两层明显地隔离，表示层的任何变化，比如是用客户端软件还是用浏览器方式表示都不会影响到业务与数据的设计。”外观小姐侃侃而谈。

“非常棒的回答，下面有请观察者小姐发言。”

“需求中提到‘要有菜单、工具栏和状态栏等要素’，其实不管是C/S架构还是B/S架构，每个按钮或链接的点击都需要触发一系列的事件。而所有的事件机制其实都是观察者模式的一种应用，即当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新，比如状态栏就是一个根据事件的不同时刻需要更新的控件。因为我觉得整个表示层，采用事件驱动的技术是可以很好地完成需求的。我说完了。”

“OK，下面有请适配器小姐回答。”

“本来我以为这个系统没我什么事，但是最后的需求，让我感觉我还是能发挥大作用。需求说‘统计图生成如若时间充裕则自行开发，否则就购买第三方的组件’，这里的意思其实就是说，统计图表的生成是自己开发还是购买组件是有变数的。既然这里存在可能的变化，那很显然要考虑将其封装，根据依赖倒转原则，我们让业务模块依赖一个抽象的生成图接口，而非具体的第三方组件或自行实现代码将有利于我们的随机应变。至于第三方组件的接口不统一的问题，完全可以利用适配器模式来处理，达到适应变化的需求。谢谢！”

“哈，看来题目难不倒各位模式的小姐。下面有请策略小姐发言。”

“轮到我了？好的，我是这么想的。公司有多种薪资发放规则，但不管用哪种发放规则，只不过是计算的不同，最终都将是数据的存储与展示，因此我们不希望发放规则的变化会影响系统的数据新增修改、数据统计查询等具体的业务逻辑。应用策略模式，可以很好的把薪资发放规则一个个封装起来，并且使它们可相互替换，这样哪怕再增加多种发放规则，或者修改原有的规则，都不会影响其他业务的实现。回答完毕。”

“Very Very Good！策略小姐的回答，非常精彩。下面有请最后一位，工厂方法小姐发言。”

“只要是在做面向对象的开发，创建对象的工作不可避免。创建对象时，负责创建的实体通常需要了解要创建的是哪个具体的对象，以及何时创建这个而非那个对象的规则。而我们如果希望遵循开放-封闭原则、依赖倒转原则和里氏代换原则，那使用对象时，就不应该知道所用

的是哪一个特选的对象。此时就需要‘对象管理者’工厂来负责此事 **[DPE]**。比如需求中说到系统做成产品后可以用多种数据库来做持久化。如果我们创建对象时指明了是用SQL Server，那么就会面临着要用Oracle的时候的尴尬问题，因为这里不能适应变化。解决办法是我们可以通过抽象工厂模式、反射技术等手段来让业务逻辑与数据访问之间减少耦合。当然，如果系统比较复杂，也可以使用一种ORM工具来将业务对象与关系数据库进行映射。同样的道理，刚才提到的应用策略模式解决薪资发放规则，用适配器模式解决第三方组件适配等问题，在对象创建时，都可以通过工厂的手段来避免指明具体对象，减少耦合性。另外，在创建对象时，使用抽象工厂、原型、建造者的设计比使用工厂方法要更灵活，但它们也更加复杂，通常，设计是以使用工厂方法开始，当设计者发现需要更大的灵活性时，设计便会向其他创建型模式演化 **[DP]**。总之，在面向对象开发过程中，为了避免耦合，都多多少少会应用工厂方法来帮助管理创建对象的工作。工厂方法的实现并不能减少工作量，但是它能够在必须处理新情况时，避免使已经很复杂的代码更加复杂 **[DPE]**。工厂方法会继续努力，为大家做好优质的服务，谢谢大家。”

“哦，我对你的敬仰真如滔滔江水、连……”*GOF*突然感觉自己有些失态，连忙收口。“下面有请6位评委慎重做出选择，评出我们本届大赛的季军、亚军，以及我们的——冠军。现在是广告时间，不要走开哦，我们马上回来揭晓答案。”

场下，只见*Hibernate*双手紧握，举在胸前，*ADO.NET*则诧异地望着他。

Hibernate：“啊，工厂方法，我真是爱死你了。”

ADO.NET：“花痴，别犯傻了，人家可是大明星了，你爱到死人家也不会知道。”

Hibernate：“我猜工厂方法准赢、确定赢、一定赢、肯定赢！”

ADO.NET：“那可难说得很。其他几位也表现很好的，主要原因是工厂方法最后一个回答，所以占了便宜，不然怎么能举出前面选手所说的例子，这其实都不太公平。”

Hibernate：“咦，你说得对哦，为什么是工厂方法最后一个发言，按道理她应该第一个发言的。”

ADO.NET：“这谁知道呢，说不定又有什么潜规则在里面。”

Hibernate：“啊，如果真是这样，那可实在是太让我这‘工仔’伤心了。”

ADO.NET：“‘公仔’？怎么听得像是广东香港那边对毛绒玩具的叫

法。这是什么意思？”

Hibernate：“你不知道呀，‘工仔’就是工厂方法的Fans的统称。”

ADO.NET：“啊，连粉丝统称都有了。太快了吧。”

此时，只听场外口号声四起。

“工厂工厂，工仔爱你，就像老鼠爱大米。”所有工仔们在简单工厂的指挥下举着条幅大叫道。

“模林至尊，策略同盟，号令天下，莫敢不从。”策略的粉丝开始齐呼。

外观的Fans坐不住了，跟着对叫道：“外观不出，谁与争锋。”

相比之下，观察者和适配器的Fans就显得没什么声势。

“各位现场的来宾，观众朋友们，第一届OOTV杯超级设计模式大赛的结果马上就要揭晓，让我们有请我们的总策划兼总导演抽象先生，上台来宣布结果。”

只见50多岁的抽象先生，缓缓走上台前，接过主持人*GOF*交给他的信封，对着话筒说：“真的很高兴能看到今天的大赛举办得圆满成功。在这我代表组办方，向所有为这次大赛付出艰苦努力的工作人员表示衷心的感谢。望着这23位模式小姐，她们在台上尽情地‘群模乱舞’，我们在台下感受到了她们的‘模法无边’。非常值得庆贺的是，23位小姐都发挥出了应有的水平，赛出了风格，赛出了水平。我……”

*GOF*打断了抽象先生的说话，“请您打开信封宣布结果。”

“哦，对对对，我是来宣布结果的。”抽象先生反应过来，拆开了信封，念道，“第一届OOTV杯超级设计模式大赛决赛入围的有：工厂方法、外观、观察者、策略、适配器。她们五位设计模式小姐表现都很出色。其实说到底，面向对象设计模式体现的就是抽象的思想，类是什么，类是对对象的抽象，抽象类呢，其实就是对类的抽象，那接口呢，说白了就是对行为的抽象。不管是什么，其实……”

“抽象先生，请您宣布结果。”*GOF*不得不再次提醒他。

“哦，你等我说完，其实都是在不同层次、不同角度进行抽象的结果，它们的共性就是抽象。所以……”抽象先生打了个咯噔，“我宣布，第一届OOTV杯超级设计模式大赛的冠军是……”

29.9　梦醒时分

时间：7月23日　晚上　23:50　　　地点：小菜自己的卧室　　　人物：小菜、大鸟

“喂，小菜，醒醒。”大鸟站在小菜旁边，推了推他，“这样趴着睡会感冒的，上床去睡。”

“啊，大鸟呀，你坏我好梦。”小菜用手揉着眼睛说道，“早不来晚不来，刚梦到要宣布结果，你就来了。气死我了。”

“你梦到什么了？都快12点了，快去睡觉吧。”

“这下是再也不可能梦到抽象宣布结果了。你说怎么办？”

“什么抽象宣布结果，你做什么鬼梦呢？”

“明天我们公司让我去做关于应用设计模式体会的演讲，我本想等你回来问问你的，一不小心就睡着了，做了一个好梦，但却被打扰了。”

“哦，这是好事呀，明天演讲什么内容，你想好了吗？”

“本来是没想好，不过现在吗，嘿嘿，我已经有数了。”

“你打算怎么讲？”

“保密，明天我回来再对你说。我去睡觉去了。”

“好吧，祝你好运。晚安。”

时间：7月24日　下午　15:30　　　地点：小菜公司大会议室　　　人物：全公司成员

“今天我给大家讲一个关于‘OOTV杯超级模式大赛’的故事。故事是这样开始的……”

小菜的故事讲得非常精彩，成了公司的技术明星。在随后的日子里，逐步完成了由菜鸟程序员向优秀软件工程师的蜕变，并继续向着成为软件架构师的理想前进。

29.10 没有结束的结尾

生活还在继续，编程也不会结束。每天晚上，小菜和大鸟继续着对程序、对爱情、对理想、对人生的讨论和思考。而我们的故事，却要暂时告一段落了。

祝愿您在阅读本书的过程当中，读有所获，阅有所思。书籍一定会有最后一页，但你的面向对象编程之路或许才刚刚开始。相信通过你的努力，你的人生会更加精彩。

附录A　培训实习生——面向对象基础

本章节主要是为阅读本书设计模式章节有困难的朋友提供的正餐前的开胃小菜。目的是不希望在阅读设计模式时，由于对C#语言中面向对象的知识了解匮乏或理解欠缺造成太大的障碍。所以本章仅对涉及阅读本书需要了解的C#语言中面向对象的知识做简单的介绍，比如像继承多态、接口抽象类、集合泛型等。由于本书的重点不是讲面向对象基础，也不是讲C#语言，所以在本章中对很多技术细节都未提及或深入，希望有兴趣的朋友去阅读相关的专著，来补充对面向对象、C#、.NET的认识不足。

A.1　培训实习生

时间：9月3日 上午 9点　　地点：小菜办公室　　人物：小菜、史熙、公司开发部经理

小菜的单位来了个大学实习生，叫史熙。开发部经理安排小菜有空教教他，小菜欣然接受。

“蔡老师，请多多关照了。”史熙很诚恳。

“哈，不敢当，我也刚毕业不久，比你大一届。以后叫我名字吧，我叫蔡遥。”

“哦，那叫遥哥。”

“嗨！不跟你客气了，你是学计算机专业的吗？”

“是的，今年大四。不过老实说，在学校里真的没学到什么东西。所谓‘公司一日，校园一年。’来这一定会比学校学的东西要多。”

“哈，夸张了，应该是‘公司一日抵自学一月。’在实践当中，自然会学得快一些。不过话说回来，一些基本的东西还是要知道的。C#学过吧，对面向对象又了解多少？”

“C#是学过的，什么变量、常量、判断循环，我都知道，我还用VS 2005写过些桌面小程序呢。面向对象，好像也学过的。什么类呀、方法呀的，太久了，当时就为了应付考试了，具体是如何用，根本不记得。你要不给我讲讲吧。”

“啊，都不记得了，不就等于白学了吗？不过没关系，今天我正好有空，我们争取来个快速入门吧。”

“太好了，遥哥，不对，还是得叫蔡老师，先谢谢了。”

A.2 类与实例

“先问问你，对象是什么？类是什么？”小菜问道。

“准确定义不知道。类大概就是对东西分类的意思。”史熙答得很勉强。

“啊，看来你是实实在在的菜鸟呀。一切事物皆为对象，即所有的东西都是对象，对象就是可以看到、感觉到、听到、触摸到、尝到、或闻到的东西。准确地说，对象是一个自包含的实体，用一组可识别的特性和行为来标识。面向对象编程，英文叫**Object-Oriented Programming**，其实就是针对对象来进行编程的意思。至于类，待会儿再讲，我们先从最简单的开始，你用VS　2005建一个Windows应用程序，最终我们将实现一个‘动物运动会’的软件小例子。”

“动物运动会？有意思。”

“首先实现这样一个功能，点击一个‘猫叫’按钮，会弹出小猫的叫声‘喵’的提示框。”

“这个非常简单。”



```
private void button1_Click(object sender,
EventArgs e)
{
    MessageBox.Show("喵");
}
```

“好了，是这个意思吗？”史熙问道。

“是，程序是编出来了，现在的问题就是，如果我们需要在另一个按钮中来让小猫叫一声，或者需要小猫多叫几声，怎么办？”

“那就多写几个‘MessageBox.Show（"喵"）;’呀。”

“那就不重复了吗？可不可以想个办法。”

“我知道你的意思，写个函数就可以解决了。其他需要‘猫叫’的地方都可以”

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show(Shout());
}

string Shout()      猫叫的方法
{
    return "喵";
}
```

“很好，现在问题是，万一别的地方，我指其他窗体也需要‘猫叫Shout（）’，如何处理？”

“这个我好像学过的，在Shout（）方法前面加一个public，别的窗体就可以访问了。”

“对，没错，这样是可以达到访问的目的了，但是你觉得这个‘Shout（猫叫）’放在这个窗体的代码中合适吗？这就好比，居委会的公用电视放在你家，而别人家都没有，于是街坊邻居都来你家看电视。你喜欢这样吗？”

“这样好呀，邻居关系好了。不过这确实不是办法，公用电视应该放在居委会。”

“所以说，这‘猫叫’的函数应该放在一个更合适的地方，这就是‘类’。类就是具有相同的属性和功能的对象的抽象的集合，我们来看代码。”

```
class Cat
{
    public string Shout()
    {
        return "喵";
    }
}
```

“这里‘class’是表示定义类的关键字，‘Cat’就是类的名称，‘Shout’就是类的方法。”

“哈，定义类这玩意还是很简单的嘛”

“这里有两点要注意，第一，类名称首字母记着要大写。多个单词则各个首字母大写；第二，对外公开的方法需要用**‘public’**修饰符。”

“明白，那怎么应用这个类呢？”

“很简单，只要将类实例化一下就可以了。”

“什么叫实例化？”

“实例，就是一个真实的对象。比如我们都是‘人’，而你和我其实就是‘人’类的实例了。而实例化就是创建对象的过程，使用**new**关键字来创建。”

```
private void button1_Click(object sender, EventArgs e)
{
    Cat cat = new Cat();          // 将 Cat 类实例化
    MessageBox.Show(cat.Shout());
}
```

“注意，‘Cat cat = new Cat();’其实做了两件事。”

```
Cat cat;              // 声明一个 Cat 的对象，对象名为 cat

cat = new Cat();      // 将此 cat 对象实例化
```

“Cat实例化后，等同于出生了一只小猫cat，此时就可以让小猫cat.Shout()了。在任何需要小猫叫的地方都可以实例化它。”

“明白，这下调用它确实就方便很多了。”

A.3 构造方法

“下面，我们希望出生的小猫应该有个姓名，比如叫‘咪咪’，当咪咪叫的时候，最好是能说‘我的名字叫咪咪，喵’。此时就需要考虑用构造方法。”

“构造方法？这是做什么用的？”

“构造方法，又叫构造函数，其实就是对类进行初始化。构造方法与类同名，无返回值，也不需要**void**，在**new**时候调用。”

“那就是说，在类创建时，就是调用构造方法的时候了？”

“是呀，在‘Cat cat＝new Cat（）;’中，new后面的Cat（）其实就是构造方法。”

“不对呀，在类当中没有写过构造方法Cat（），怎么可以调用呢？”

“问得好，实际情况是这样，所有类都有构造方法，如果你不编码则系统默认生成空的构造方法，若你有定义的构造方法，那么默认的构造方法就会失效了。也就是说，由于你没有在Cat类中定义过构造方法，所以C#语言会生成一个空的构造方法Cat（）。当然，这个空的方法是什么也不做，只是为了让你能顺利地实例化而已。”

“那不是很好吗？我们还需要构造方法做什么呢？”

“刚才不是说过吗，构造方法是为了对类进行初始化。比如我们希望每个小猫一诞生就有姓名，那么就应该写一个有参数的构造方法。”

```
class Cat
{
    private string name = "";
    public Cat(string name)
    {
        this.name = name;
    }

    public string Shout()
    {
        return "我的名字叫"+name+" 喵";
    }
}
```

声明 Cat 类的私有字符串变量 name

定义 Cat 类的构造方法，参数是输入一个字符串

将参数赋值给私有变量 name

“这样一来，我们在客户端要生成小猫时，就必须要给小猫起名字了。”

```
private void button1_Click(object sender,
EventArgs e)
{
    Cat cat = new Cat("咪咪");
    MessageBox.Show(cat.Shout());
}
```

运行效果如下



A.4 方法重载

“但是，遥哥，如果我事先没有起好小猫的名字，难道这个实例就创建不了了吗？”史熙又有疑问。

“是的，有些父母刚生下孩子时，姓名没有起好也是很正常的事。就目前的代码，你如果写‘Cat cat = new Cat ( ) ;’是会直接报‘Cat方法没有采用0个参数的重载’的错误，原因就是必须要给小猫起名字。如果当真需要不起名字也要生出小猫来。可以用‘方法重载’。”

“方法重载？好像也学过的东西，具体如何说？”

“方法重载提供了创建同名的多个方法的能力，但这些方法需使用不同的参数类型。注意并不是只有构造方法可以重载，普通方法也是可以重载的。”

```
class Cat
{
    private string name = "";
    public Cat(string name)
    {
        this.name = name;
    }

    public Cat()            // 将构造方法重载
    {
        this.name = "无名";
    }

    public string Shout()
    {
        return "我的名字叫"+name+" 喵";
    }
}
```

“哦，这样的话，如果写‘Cat cat = new Cat ( ) ;’的话，就不会报错了。而猫叫时会是‘我的名字叫无名喵’。”

“对的，注意方法重载时，两个方法必须要方法名相同，但参数类型或个数必须要有所不同，否则重载就没有意义了。你觉得方法重载的好处是什么？”小菜问道。

“哈，我想应该是方法重载可在不改变原方法的基础上，新增功能。”

“说得很好，方法重载算是提供了函数可扩展的能力。比如刚才这个例子，有的小猫起好名字了，就用带string参数的构造方法，有的没有

名字，就用不带参数的，这样就达到了扩展的目的。”

“如果我需要分清楚猫的姓和名，还可以再重载一个public Cat（string firstName，string lastName），对吧？”

“对的。非常好。下面，我们觉得小猫叫的次数太少，希望是我让它叫几声，它就叫几声，如何做？”

“那是不是在构造方法里再加一个叫的次数？”

“那样当然是可以，但叫几声并不是必须要实例化的时候就声明的，我们可以之后再规定叫几声，所以这时应该考虑用‘属性’。”

A.5 属性与修饰符

“属性是一个方法或一对方法，但在调用它的代码看来，它是一个字段，即属性适合于以字段的方式使用方法调用的场合。这里还需要解释一下字段的意思，字段是存储类要满足其设计所需要的数据，字段是与类相关的变量。比如刚才的Cat类中的‘private string name = "";’name其实就是一个字段，它通常是私有的类变量。那么属性是什么样呢？我们现在增加一个‘猫叫次数ShoutNum’的属性。”

```
private int shoutNum = 3;
public int ShoutNum
{
    get
    {
        return shoutNum;
    }
    set
    {
        shoutNum = value;
    }
}
```


“刚才没有强调public和private的区别，它们都是修饰符，**public**表示它所修饰的类成员可以允许其他任何类来访问，俗称公有的。而**private**表示只允许同一个类中的成员访问，其他类包括它的子类无法访问，俗称私有的。如果在类中的成员没有加修饰符，则被认为是**private**的。修饰符还有其他三个，我们以后再讲。通常字段都是**private**，即私有的变量，而属性都是**public**，即公有的变量。那么在这里shoutNum就是私有的字段，而ShoutNum就是公有的对外属性。由于是对外的，所以属性的名称一般首字母大写，而字段则一般首字母小写或前加‘_’。”

“属性的get和set是什么意思？”

“属性有两个方法**get**和**set**。**get**访问器返回与声明的属性相同的数据类型，表示的意思是调用时可以得到内部字段的值或引用；**set**访问器没有显式设置参数，但它有一个隐式参数，用关键字**value**表示，它的作用是调用属性时可以给内部的字段或引用赋值。”

“那又何必呢，我把字段的修饰符改为public，不就可以做到对变量的既读又写了吗？”

“是的，如果仅仅是可读可写，那与声明了public的字段没什么区别。但是对于对外界公开的数据，我们通常希望能做更多的控制，这就好像我们的房子，我们并不希望房子是全透明的，那样你在家里的所有活动全部都被看得清清楚楚，毫无隐私可言。通常我们的房子有门有窗，但更多的是不透明的墙。这门和窗其实就是public，而房子内的东

西，其实就是private。而对于这个房子来说，门窗是可以控制的，我们并不是让所有的人都可以从门随意进出，也不希望蚊子苍蝇来回出入。这就是属性的作用了，如果你把字段声明为public，那就意味着不设防的门窗，任何时候，调用者都可以读取或写入，这是非常糟糕的一件事。如果把对外的数据写成属性，那情况就会好很多。”

```
private int shoutNum = 3;
public int ShoutNum
{
    get
    {
        return shoutNum;
    }
}
```

去掉了 set，表示 ShoutNum 属性是只读的

```
private int shoutNum = 3;
public int ShoutNum
{
    get
    {
        return shoutNum;
    }
    set
    {
        if (value<=10)
            shoutNum = value;
        else
            shoutNum = 10;
    }
}
```

控制叫声次数，最多只能叫 10 声

“我明白了，这就好比给窗子安装了纱窗，只让阳光和空气进入，而蚊子苍蝇就隔离。多了层控制就多了层保护。”

“说得很好。我们还没有做完，由于有了‘叫声次数’的属性，于是我们的Shout方法就需要改进了。”

```
public string Shout()
{
    string result = "";
    for (int i = 0; i < shoutNum; i++)
    {
        result += "喵  ";
    }
    return "我的名字叫" + name + " " + result;
}
```

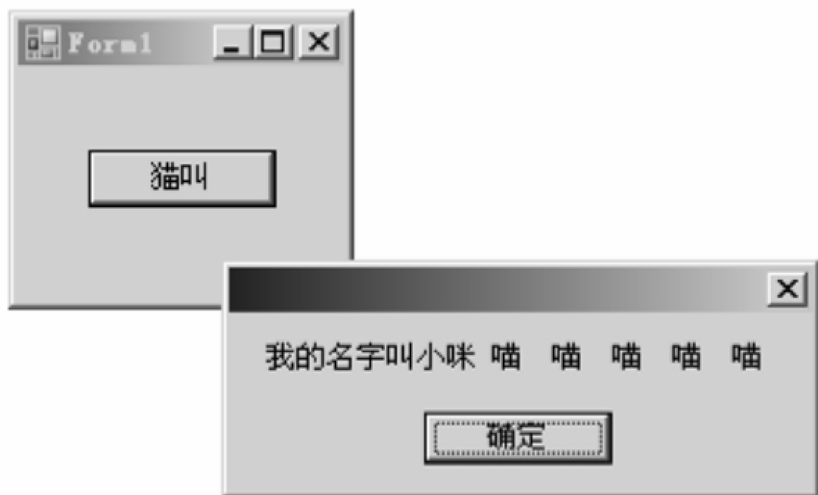
做一个循环，以便让小猫叫相应的次数

“此时调用的时候只需要给属性赋值就可以了。”

```
private void button1_Click(object sender, EventArgs e)
{
    Cat cat = new Cat("小咪");
    cat.ShoutNum = 5;
    MessageBox.Show(cat.Shout());
}
```

给属性赋值

显示结果



“如果我们不给属性赋值，小猫会叫‘喵’吗？”

“当然会，应该是三声吧，因为字段shoutNum的初始值是3。”

“很好。另外需要强调的是，变量私有的叫字段，公有的是属性，那么对于方法而言，同样也就有私有方法和公有方法了，一般无需要对外界公开的方法都应该设置其修饰符为private（私有）。这才有利于‘封装’”

A.6　封装

“现在我们可以讲面向对象的三大特性之一‘封装’了。每个对象都包含它能进行操作所需要的所有信息，这个特性称为封装，因此对象不必依赖其他对象来完成自己的操作。这样方法和属性包装在类中，通过类的实例来实现。”

“是不是刚才提炼出Cat类，其实就是在做封装？”

“是呀，封装有很多好处，第一、良好的封装能够减少耦合，至少我们让Cat和Form1的耦合分离了。第二、类内部的实现可以自由地修改，这也是显而易见的，我们已经对Cat做了很大的改动。第三、类具有清晰的对外接口，这其实指的就是定义为public的ShoutNum属性和Shout方法。”

“封装的好处很好理解呀，比如刚才举的例子。我们的房子就是一个类的实例，室内的装饰与摆设只能被室内的居住者欣赏与使用，如果没有四面墙的遮挡，室内的所有活动在外人面前一览无遗。由于有了封装，房屋内的所有摆设都可以随意地改变而不用影响他人。然而，如果没有门窗，一个包裹得严严实实的黑箱子，即使它的空间再宽阔，也没有实用价值。房屋的门窗，就是封装对象暴露在外的属性和方法，专门供人进出，以及流通空气、带来阳光。

“现在我需要增加一个狗叫的功能，就是加一个按钮‘狗叫’，点击后会弹出‘我的名字叫XX 汪 汪汪’如何做？”

“那简单呀，仿造Cat加一个Dog类就可以了。然后再增加一个button2按钮，写上click事件代码。”

```
private void button2_Click(object sender,
EventArgs e)
{
     Dog dog = new Dog("旺财");
    dog.ShoutNum = 5;
     MessageBox.Show(dog.Shout());
}
```

显示结果



“这下就OK了，小狗旺财也会叫了。”

“很好，但你有没有发现，Cat和Dog有非常类似的代码？”

“是呀，90%的代码是一样的，不过这些代码都是必须的，也没什么办法去除呀。”

“当然可以想办法，代码有大量重复不会是什么好事情。我们要用到面向对象的第二大特性‘继承’。”

A.7 继承

“我们还是先离开软件编程，来想想我们的动物常识，猫和狗都是什么？”小菜问道。

“都是给人添麻烦的东西。除了吃喝拉撒睡，什么也不干的家伙。”史熙调皮地答道。

“拜托，正经一些。猫和狗都是动物，准确地说，他们都是哺乳动物。哺乳动物有什么特征？”

“哦，这个小时候学过，哺乳动物是胎生、哺乳、恒温的动物。”

“OK，因为猫和狗是哺乳动物，所以猫和狗就同样具备胎生、哺乳、恒温的特征。所以我们可以这样说，由于猫和狗是哺乳动物，所以猫和狗与哺乳动物是继承关系。”

“哦，原来继承就是这个意思。”

“是的，回到编程上，对象的继承代表了一种**‘is-a’**的关系，如果两个对象**A**和**B**，可以描述为**‘B**是**A’**，则表明**B**可以继承**A**。‘猫是哺乳动物’，就说明了猫与哺乳动物之间继承与被继承的关系。实际上，继承者还可以理解为是对被继承者的特殊化，因为它除了具备被继承者的特性外，还具备自己独有的个性。例如，猫就可能拥有抓老鼠、爬树等‘哺乳动物’对象所不具备的属性。因而，在继承关系中，继承者可以完全替换被继承者，反之则不成立。所以，我们在描述继承的‘is-a’关系时，是不能相互颠倒的。说‘哺乳动物是猫’显然有些莫名其妙。继承定义了类如何相互关联，共享特性。继承的工作方式是，定义父类和子类，或叫做基类和派生类，其中子类继承父类的所有特性。子类不但继承了父类的所有特性，还可以定义新的特性。”

“‘is-a’这个比较好理解。”

“学习继承最好是记住三句话，如果子类继承于父类，第一、子类拥有父类非**private**的属性和功能；第二、子类具有自己的属性和功能，即子类可以扩展父类没有的属性和功能；第三、子类还可以以自己的方式实现父类的功能（方法重写）。”

“这里有些不理解，什么叫非private，难道除了public还有别的修饰符吗？”

“当然有，刚才讲了private和public，现在再讲一个protected修饰符。**protected**表示继承时子类可以对基类有完全访问权，也就是说，用protected修饰的类成员，对子类公开，但不对其他类公开。所以子类

继承于父类，则子类就拥有了父类的除private外的属性和功能，注意除这三个修饰符外还有两个，由于和目前所讲的内容无关，就留给你自己去查MSDN看吧。”

“那方法重写是什么意思？”

“这个留到后面讲多态的时候去说，现在我们来看看怎么做。对比观察Cat和Dog类。”

```
class Cat
{
    private string name = "";
    public Cat(string name)
    {
        this.name = name;
    }

    public Cat()
    {
        this.name = "无名";
    }

    private int shoutNum = 3;
    public int ShoutNum
    {
        get
        {
            return shoutNum;
        }
        set
        {
            shoutNum = value;
        }
    }

    public string Shout()
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
        {
            result += "喵  ";
        }
        return "我的名字叫" + name + " " +
        result;
    }
}
```

```
class Dog
{
    private string name = "";
    public Dog(string name)
    {
        this.name = name;
    }

    public Dog()
    {
        this.name = "无名";
    }

    private int shoutNum = 3;
    public int ShoutNum
    {
        get
        {
            return shoutNum;
        }
        set
        {
            shoutNum = value;
        }
    }

    public string Shout()
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
        {
            result += "汪  ";
        }
        return "我的名字叫" + name + " " +
        result;
    }
}
```

“我们会发现大部分代码都是相同的，所以我们现在建立一个父类，动物Animal类，显然猫和狗都是动物。我们把相同的代码尽量放到动物类中。”

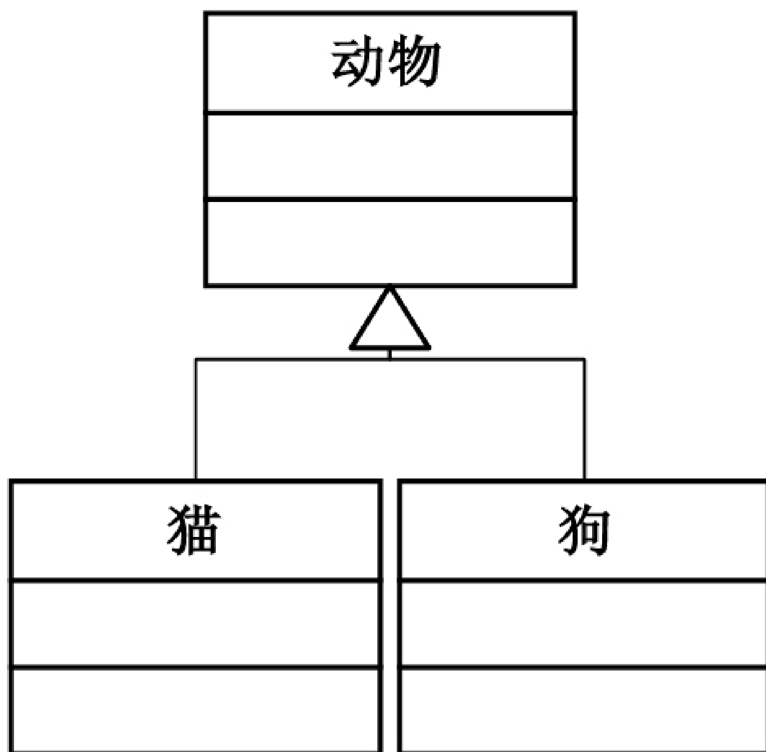
```
class Animal
{
    protected string name = "";
    public Animal(string name)
    {
        this.name = name;
    }
    public Animal()
    {
        this.name = "无名";
    }

    protected int shoutNum = 3;
    public int ShoutNum
    {
        get
        {
            return shoutNum;
        }
        set
        {
            shoutNum = value;
        }
    }
}
```

注意修饰符改为了 protected

注意修饰符改为了 protected

“然后我们需要写Cat和Dog的代码。让它们继承Animal。这样重复的部分都可以不用写了，不过在C#中，子类从它的父类中继承的成员有方法、域、属性、事件、索引指示器，但对于构造方法，有一些特殊，它不能被继承，只能被调用。对于调用父类的成员，可以用**base**关键字。”



继承格式就是子类 ： 父类

子类构造方法需要调用父类同样参数类型的构造方法，用 base 关键字代表父类

```
class Cat : Animal
{
    public Cat() : base()
    { }

    public Cat(string name) : base(name)
    { }

    public string Shout()
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
            result += "喵，";

        return "我的名字叫" + name
                + " " + result;
    }
}
```

```
class Dog : Animal
{
    public Dog() : base()
    { }

    public Dog(string name) : base(name)
    { }

    public string Shout()
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
            result += "汪，";

        return "我的名字叫" + name
                + " " + result;
    }
}
```

“此时客户端的代码完全一样，没有受到影响，但重复的代码却因此减少了。”小菜说。

“差不太多嘛，子类还是有些复杂，没简单到哪去？”史熙说道。

“如果现在需要加牛、羊、猪等多个类似的类，按你以前的写法就需要再复制三遍，也就是有五个类。如果我们需要改动起始的叫声次数，也就是让shoutNum由3改为5，你需要改几个类？”

“我懂你意思了，那需要改5个类，现在有了Animal，就只要改一个类就行了，继承可以使得重复减少。”

“说得很好。不用继承的话，如果要修改功能，就必须在所有重复的方法中修改，代码越多，出错的可能就越大，而继承的优点是，继承使得所有子类公共的部分都放在了父类，使得代码得到了共享，这就避免了重复，另外，继承可使得修改或扩展继承而来的实现都较为容易。”

“嗯，继承是个好东西，我以前时常Ctrl＋C加Ctrl＋V的，这样表面上很快，但其实重复的代码越多，以后要更改的难度越大。看来以后我要多多用继承。”

“慢，等等，你说以后要多多用继承？那可不一定是好事，用是可以，但一定要慎重。”

“有这么严重吗，反正只要是有重复的时候，就继承一个子类不就行了吗？”

“哈，那真是大错特错了，那我问你，你先写了Cat，而后要你写一个Dog，由于都差不多，你有没有考虑过直接让狗去继承猫呢？”

“咦，这其实也差不多哦，没什么问题呀。如果再有了羊呀、牛呀也都分别继承猫就可以了。”

“问题就在这里了。这就使得猫会的行为，狗都会。现在编写的这只猫只会叫，以后它应该还可以会抓老鼠、爬树等行为，你让狗继承了猫，就意味着狗也就会抓老鼠、会爬树。你觉得这合理吗？”

“狗拿耗子，那是多管闲事了。看来不能让狗继承猫，那样很容易造成不必要的麻烦。”

“继承是有缺点的，那就是父类变，则子类不得不变。让狗去继承于猫，显然不是什么好的设计，另外，继承会破坏包装，父类实现细节暴露给子类，这其实是增大了两个类之间的耦合性。”

“什么叫耦合性？”

“严格定义你自己去查吧，简单理解就是藕断丝连，两个类尽管分开，但如果关系密切，一方的变化都会影响到另一方，这就是耦合性高的表现，继承显然是一种类与类之间强耦合的关系。”

“明白，你说了这么多，那什么时候用继承才是合理的呢？”

“我最先不是说过吗？当两个类之间具备**‘is-a’**的关系时，就可以考虑用继承了，因为这表示一个类是另一个类的特殊种类，而当两个类之间是‘has-a’的关系时，表示某个角色具有某一项责任，此时不合适用继承。比如人有两只手，手不能继承人；再比如飞机场有飞机，这飞机也不能去继承飞机场。”

“OK，也就是说，只有合理的应用继承才能发挥好的作用。”

A.8　多态

“下面我们再来增加需求，如果我们要举办一个动物运动会，来参加的有各种各样的动物，其中有一项是‘叫声比赛’。界面就是放两个按钮，一个是‘动物报名’，就是确定动物的种类和报名的顺序，另一个是‘叫声比赛’，就是报名的动物挨个地叫出声音来比赛。注意来报名的都是什么动物，我们并不知道。可能是猫、可能是狗，也可能是其他的动物，当然它们都需要会叫才行。”

“有点复杂，我除了会加两个按钮外，不知道如何做了。”

“先分析一下，来报名的都是动物，参加叫声比赛必须会叫。这说明什么？”

“说明都有叫的方法，哦，也就是Animal类中有Shout方法。”

“是呀，所谓的‘动物报名’，其实就是建立一个动物对象数组，让不同的动物对象加入其中。所谓的‘叫声比赛’，其实就是遍历这个数组来让动物们‘Shout（ ）’就可以了。”

“哦，我大概明白你意思了，那看看我写的代码，我觉得应该是类似的样子，但问题是我们不知道是哪个动物来报名，最终叫的时候到底是猫在叫还是狗在叫呢？”

```
private Animal[] arrayAnimal;                    // 声明一个动物数组

//“动物报名”的按钮事件
private void button3_Click(object sender, EventArgs e)
{
    arrayAnimal = new Animal[5];                 // 实例化最多可报名5个的动物数组对象

        ?                                        // 谁来报名我不知道？
}

//“叫声比赛”的按钮事件
private void button4_Click(object sender, EventArgs e)

{
    foreach (Animal item in arrayAnimal)         // 遍历这个数组，让它们都Shout()
    {
        MessageBox.Show(item.Shout());           // 是什么动物在叫我也不知道？
    }
}
```

“是呀，就之前讲到的知识，是不足以解决这个问题的，所以我们引

入面向对象的第三大特性——多态。”

“啊，多态，多态是我大学里听得很多，但一直都不懂的东西，实在不明白它是什么意思。”

“多态表示不同的对象可以执行相同的动作，但要通过它们自己的实现代码来执行。看定义显然不太明白，我来举个例子你就好理解了。我们的国粹‘京剧’以前都是子承父业，代代相传的艺术。假设有这样一对父子，父亲是非常有名的京剧艺术家，儿子长大成人，模仿父亲的戏也惟妙惟肖。有一天，父亲突然发高烧，上不了台表演，而票都早就卖出，退票显然会大大影响声誉。怎么办呢？由于京戏都是需要化妆才可以上台的，于是就决定让儿子代父亲上台表演。”

“化妆后谁认识谁呀，只要唱得好就可以糊弄过去了。”

“是呀，这里面有几点注意，第一，子类以父类的身份出现，儿子代表老子表演，化妆后就是以父亲身份出现了。第二、子类在工作时以自己的方式来实现，儿子模仿得再好，那也是模仿，儿子只能用自己理解的表现方式去模仿父亲的作品；第三、子类以父类的身份出现时，子类特有的属性和方法不可以使用，儿子经过多年学习，其实已经有了自己的创作，自己的绝活，但在此时，代表父亲表演时，绝活是不能表现出来的。当然，如果父亲还有别的儿子会表演，也可以在此时代表父亲上场，道理也是一样的。这就是多态。”

“听听好像都懂，怎么用呢？”

“是呀，怎么用呢，我们还需要了解一些概念，虚方法和方法重写。为了使子类的实例完全接替来自父类的类成员，父类必须将该成员声明为虚拟的。这是通过在该成员的返回类型之前添加**virtual**关键字来实现。通常虚拟的是方法，但其实除了字段不能是虚拟的，属性、事件和索引器都可以是虚拟的。尽管方法可以是虚拟的，但虚方法还是有方法体，可以实际做些事情。然后，子类可以选择使用**override**关键字，将父类实现替换为它自己的实现，这就是方法重写**Override**，或者叫做方法覆写。我们来看一下例子。”

“由于Cat和Dog都有Shout的方法，只是叫的声音不同，所以我们可以让Animal有一个Shout的虚方法，然后Cat和Dog去重写这个Shout，用的时候，就可以用猫或狗代替Animal叫唤，来达到多态的目的。”

```
class Animal
{
    ......
    public virtual string Shout()
    {
        return "";
    }
}
```

注意修饰符中增加了一个 virtual，它表示此方法是虚方法，可以被子类重写

```
class Cat : Animal
{
    public Cat() : base()
    { }

    public Cat(string name) : base(name)
    { }

    public override string Shout()
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
            result += "喵，";

        return "我的名字叫" + name
                + " " + result;
    }
}
```

增加了 override，表示方法重写

```
class Dog : Animal
{
    public Dog() : base()
    { }

    public Dog(string name) : base(name)
    { }

    public override string Shout()
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
            result += "汪，";

        return "我的名字叫" + name
                + " " + result;
    }
}
```

“再回到你刚才写的客户端代码上。”

```
private Animal[] arrayAnimal;

//“动物报名”的按钮事件
private void button3_Click(object sender, EventArgs e)
{
    arrayAnimal = new Animal[5];
    arrayAnimal[0] = new Cat("小花");
    arrayAnimal[1] = new Dog("阿毛");
    arrayAnimal[2] = new Dog("小黑");
    arrayAnimal[3] = new Cat("娇娇");
    arrayAnimal[4] = new Cat("咪咪");
}

//“叫声比赛”的按钮事件
private void button4_Click(object sender, EventArgs e)
{
    foreach (Animal item in arrayAnimal)
    {
        MessageBox.Show(item.Shout());
    }
}
```

声明一个动物数组

实例化最多可报名 5 个的动物数组对象

报名的分别是猫、狗、狗、猫、猫

遍历这个数组，让它们都 Shout()

由于有了多态性，所以叫的时候，程序会自动去找 item 是什么对象，然后用哪个重写方法

结果显示，先点击“动物报名”，然后“叫声比赛”，将有五个提示框依次弹出。



“我明白了，Animal相当于京剧表演的老爸，Cat和Dog相当于儿子，儿子代表父亲表演Shout，但Cat叫出来的是‘喵’，Dog叫出来的是‘汪’，这就是所谓的不同的对象可以执行相同的动作，但要通过它们自己的实现代码来执行。”

“说得好，是这个意思，不过一定要注意了，这个对象的声明必须是父类，而不是子类，实例化的对象是子类，这才能实现多态。多态的原理是当方法被调用时，无论对象是否被转换为其父类，都只有位于对象继承链最末端的方法实现会被调用。也就是说，虚方法是按照其运行时类型而非编译时类型进行动态绑定调用的。[AMNFP]”

```
动物 animal = new 猫();
```

或者

```
猫 cat = new 猫();
动物 animal = cat;
```

“不过老实说，即使这样，我也还是不太理解这样做有多大的好处。多态被称为面向对象三大特性，我感觉不到它有和封装、继承同样的作

用。”

“慢慢来，要深刻理解并会合理利用多态，不去研究设计模式是很难做到的。也可以反过来说，没有学过设计模式，那么对多态、乃至对面向对象的理解多半都是肤浅和片面的。我相信会有那种天才，可以听一知十，刚学的东西就可以灵活自如地应用，甚至要造汽车，他都能再去发明轮子。但对于绝大多数程序员，还是需要踏踏实实地学习基本的东西，并在不断地实践中成长，最终成为高手。”

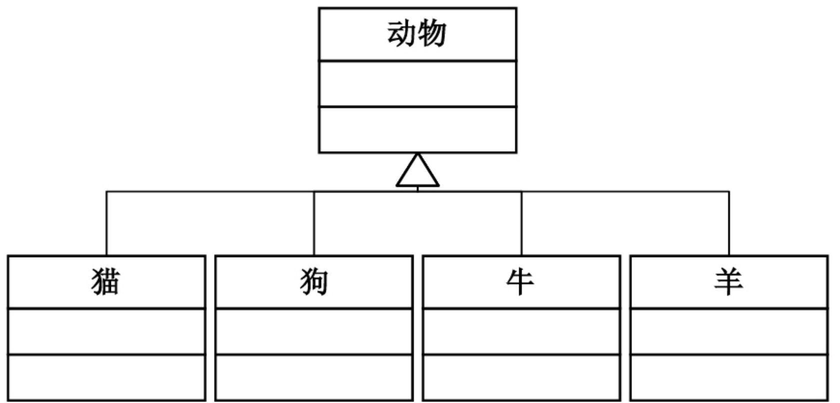
“蔡老师，受教了。下面我们做什么？”

（注：多态中样例改编自《编程的奥秘》，作者金旭亮，电子工业出版社出版。）

A.9 重构

“现在又来了小牛和小羊来报名，需要参加‘叫声比赛’，你如何做？”

“这个简单了，我现在再实现牛Cattle和羊Sheep的类，让它们都继承Animal就可以了。”




```
class Cattle : Animal
{
    public Cattle () : base()
    { }

    public Cattle (string name):base(name)
    { }

    public override string Shout()
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
            result += "哞，";

        return "我的名字叫" + name
                + " " + result;
    }
}
```

```
class Sheep : Animal
{
    public Sheep () : base()
    { }

    public Sheep (string name) : base(name)
    { }

    public override string Shout()
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
            result += "咩，";

        return "我的名字叫" + name
                + " " + result;
    }
}
```

“等等，你有没有发现，猫狗牛羊四个类，除了构造方法之外，还有重复的地方？”

“是呀，我发现了，Shout里除了四种动物叫的声音不同外，几乎没有任何差异。”

“这有什么坏处？”

“重复呀，如果你有需求说，把‘我的名字叫XXX’改成‘我叫XXX’，我

就得更改四个类的代码了。”

“非常好，所以这里有重复，我们还是应该要改造它。”

“我先把重复的这个Shout的方法体放到Animal类中，去掉virtual。”

“这样如何能行，动物叫什么声音呢？叫‘喵’？叫‘汪’？都不行，动物是个抽象的概念，它是不会有叫的声音的。”

“别急，我们把叫的声音部分改成另一个方法getShoutSound不就行了！”

```
class Animal
{
    ......

    public string Shout()          // 去除 virtual，成为普通的公共方法
    {
        string result = "";
        for (int i = 0; i < shoutNum; i++)
            result += getShoutSound() + ", ";   // 此处是原先子类的唯一不同之处，所以改成调用一个虚方法getShoutSound

        return "我的名字叫" + name + " " + result;
    }
    protected virtual string getShoutSound()   // “得到叫声”，虚方法，让子类重写，只需给继承的子类使用，所以用protected 修饰符
    {
        return "";
    }
}
```

“此时的子类就极其简单了。除了叫声和构造方法的不同，所有的重复都转移到了父类，真是漂亮之极。”

```
class Cat : Animal
{
   public Cat () : base()
   { }

   public Cat (string name) : base(name)
   { }

   protected override string getShoutSound ()
   {
      return "喵";
   }
}
```

```
class Dog : Animal
{
   public Dog () : base()
   { }

   public Dog (string name) : base(name)
   { }

   protected override string getShoutSound ()
   {
      return "汪";
   }
}
```

```
class Cattle : Animal
{
   public Cattle () : base()
   { }

   public Cattle (string name):base(name)
   { }

   protected override string getShoutSound ()
   {
      return "哞";
   }
}
```

```
class Sheep : Animal
{
   public Sheep () : base()
   { }

   public Sheep (string name) : base(name)
   { }

   protected override string getShoutSound()
   {
      return "咩";
   }
}
```

“有点疑问，这样改动，子类，比如Cat就没有Shout方法了，外面如何调用呢？”

“嗨，你把继承的基本忘记了？继承第一条是什么？”

“哈，是子类拥有所有父类非private的属性和方法。对的对的，由于子类继承父类，所以是public的Shout方法是一定可以为所有子类所用的。”史熙高兴地说，“我渐渐能感受到面向对象编程的魅力了，的确是非常的简捷。由于不重复，所以需求的更改都不会影响到其他类。”

“这里其实就是在用一个设计模式，叫模板方法。（详见第10章）”

“啊，原来就是设计模式呀，Very Good，太棒了，哈，我竟然学会了设计模式。”

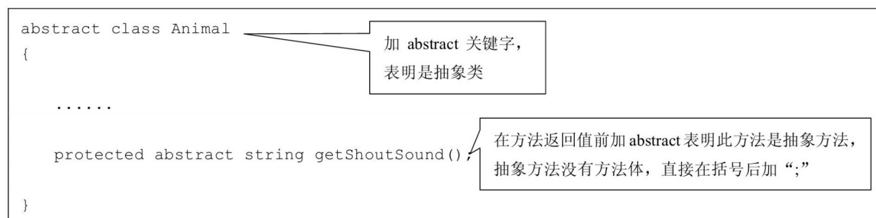
“疯了？发什么神经呀。”小菜同样微笑道，“这才是知道了皮毛，得意什么，还早着呢。”

A.10　抽象类

“我们再来观察，你会发现，Animal类其实根本就不可能实例化的，你想呀，说一只猫长什么样，可以想象，说new Animal（）；即实例化一个动物。一个动物长什么样？”

“不知道，动物是一个抽象的名词，没有具体对象与之对应。”

“是呀，所以我们完全可以考虑把实例化没有任何意义的父类，改成抽象类，同样的，对于Animal类的getShoutSound方法，其实方法体没有任何意义，所以可以将virtual修饰符改为abstract，使之成为抽象方法。C#允许把类和方法声明为**abstract**，即抽象类和抽象方法。”



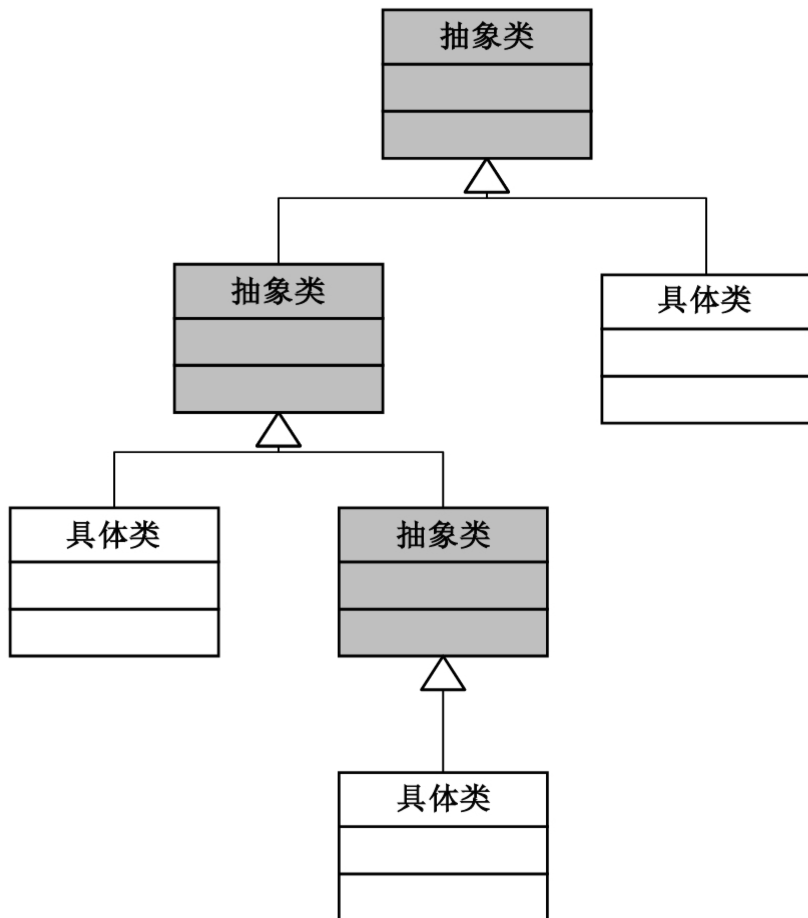
“这样一来，Animal就成了抽象类了。抽象类需要注意几点，第一，抽象类不能实例化，刚才就说过，‘动物’实例化是没有意义的；第二，抽象方法是必须被子类重写的方法，不重写的话，它的存在又有什么意义呢？其实抽象方法可以被看成是没有实现体的虚方法；第三，如果类中包含抽象方法，那么类就必须定义为抽象类，不论是否还包含其他一般方法。”

“这么说的话，一开始就可以把Animal类设成抽象类了，根本没有必要存在虚方法的父类。是这样吧？”史熙问道。

“的确是这样，我们应该考虑让抽象类拥有尽可能多的共同代码，拥有尽可能少的数据[J&DP]。”

“那到底什么时候应该用抽象类呢？”

“抽象类通常代表一个抽象概念，它提供一个继承的出发点，当设计一个新的抽象类时，一定是用来继承的，所以，在一个以继承关系形成的等级结构里面，树叶节点应当是具体类，而树枝节点均应当是抽象类[J&DP]。也就是说，具体类不是用来继承的。我们作为编程设计者，应该要努力做到这一点。比如，若猫、狗、牛、羊是最后一级，那么它们就是具体类，但如果还有更下面一级的金丝猫继承于猫、哈巴狗继承于狗，就需要考虑把猫和狗改成抽象类了，当然这也是需要具体情况具体分析的。”



“这个应该可以理解。”

“OK，我们继续下面的需求实现。”

A.11　接口

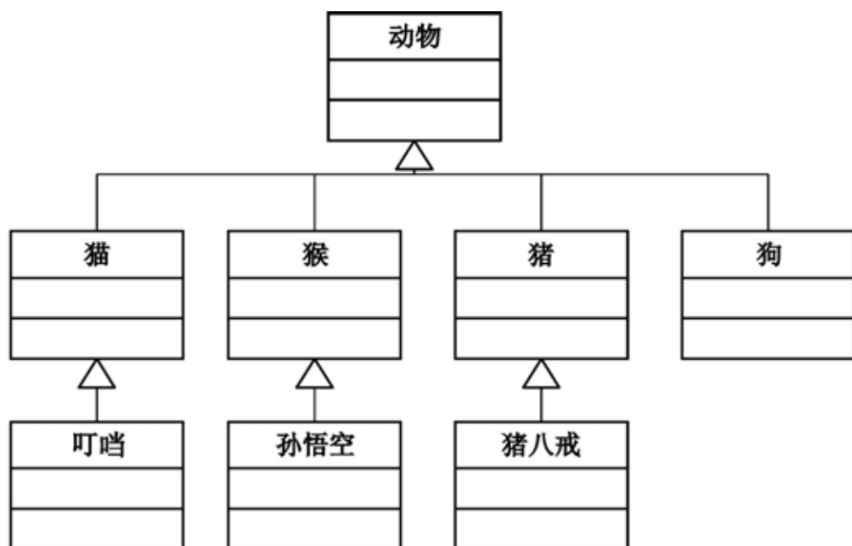
“在动物运动会里还有一项非常特殊的比赛是为了给有特异功能的动物展示其特殊才能的。”

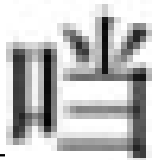
“哈，有特异功能？有意思。不知是什么动物？”

“多的是呀，可以来比赛的比如有机器猫叮当，石猴孙悟空，肥猪猪八戒，再比如蜘蛛人、蝙蝠侠等。”

“啊，这都是什么动物呀，根本就是人们虚构之物。”

“让猫狗比赛叫声难道就不是虚构？你当它们会愿意相互攀比？其实我的目的只是为了让两个动物尽量的不相干而已。现在叮当会从肚皮的口袋里变出东西，而孙悟空可以拔根毫毛变出东西，且有七十二般变化，八戒有三十六般变化。它们各属于猫、猴、猪，现在需要让它们比赛谁变东西的本领大。你来分析一下如何做？”



“‘变出东西’应该是叮、孙悟空、猪八戒的行为方法，要想用多态，就得让猫、猴、猪有‘变出东西’的能力，而为了更具有普遍意义，干脆让动物具有‘变出东西’的行为，这样就可以使用多态了。”

“哈哈，史熙呀，你犯了几乎所有学面向对象的人都会犯的错误，‘变出东西’它是动物的方法吗？如果是，那是不是所有的动物都必须具备‘变出东西’的能力呢？”

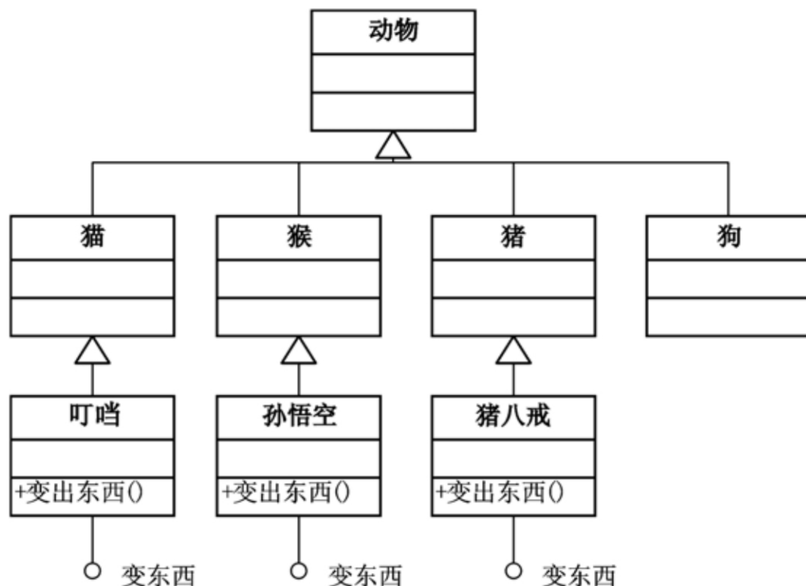
“这个，确实不是，这其实只是三个特殊动物具备的方法。那应该如何做？”

“这时候我们就需要新的知识了，那就是接口interface。接口是把隐式公共方法和属性组合起来，以封装特定功能的一个集合。一旦类实现了接口，类就可以支持接口所指定的所有属性和成员。声明接口在语法上与声明抽象类完全相同，但不允许提供接口中任何成员的执行方式。所以接口不能实例化，不能有构造方法和字段；不能有修饰符，比如public、private等；不能声明虚拟的或静态的等。还有实现接口的类就必须要实现接口中的所有方法和属性。”

“怎么接口这么麻烦。”

“要求是多了点，但一个类可以支持多个接口，多个类也可以支持相同的接口。所以接口的概念可以让用户和其他开发人员更容易理解其他人的代码。哦，对了，记住，接口的命名，前面要加一个大写字母‘I’，这是规范。”

“听不懂呀，不如讲讲实例吧。”



“我们先创建一个接口，它是用来‘变东西’用的。注意接口用**interface**声明，而不是**class**，接口名称前要加‘**I**’，接口中的方法或属性前面不能有修饰符、方法没有方法体。”

```
interface IChange
{
    string ChangeThing(string thing);
}
```

声明一IChange接口，此接口有一个方法ChangeThing，参数是一个字符串变量，返回一字符串

“然后我们来创建机器猫的类。”

```
class MachineCat:Cat,IChange
{
    public MachineCat()
        : base()
    {
    }

    public MachineCat(string name)
        : base(name)
    {
    }

    public string ChangeThing(string thing)
    {
        return base.Shout() + " 我有万能的口袋，我可变出：" + thing;
    }
}
```

机器猫继承于猫，并实现 IChange 接口，注意 Cat 与 IChange 是用“,”分隔

实现接口的方法，注意不能加 override 修饰符

base.Shout()表示调用父类Cat的方法

“猴子的类Monkey和孙悟空的类StoneMonkey与上面非常类似，在

此省略。此时我们的客户端，可以加一个‘变出东西’按钮，并实现下面的代码。”

```
private void button6_Click(object sender, EventArgs e)
{
    MachineCat mcat = new MachineCat("叮当");
    StoneMonkey wukong = new StoneMonkey("孙悟空");

    IChange[] array = new IChange[2];
    array[0] = mcat;
    array[1] = wukong;

    MessageBox.Show(array[0].ChangeThing("各种各样的东西！"));
    MessageBox.Show(array[1].ChangeThing("各种各样的东西！"));
}
```

创建两个类的实例

声明一个接口数组。将两个类实例赋值给数组

利用多态性，实现不同的 ChangeThing

运行结果



“哦，我明白了，其实这和抽象类是很类似的，由于我现在要让两个完全不相干的对象，叮和孙悟空来做同样的事情‘变出东西’，所以我不得不让他们去实现做这件‘变出东西’的接口，这样的话，当我调用接口的‘变出东西’的方法时，程序就会根据我实现接口的对象来做出反应，如果是叮，就是用万能口袋，如果是孙悟空，就是七十二变，利用了多态性完成了两个不同的对象本不可能完成的 任务。”

“说得非常好，同样是飞，鸟用翅膀飞，飞机用引擎加机翼飞，而超人呢？举起两手，握紧拳头就能飞，它们是完全不同的对象，但是，如果硬要把它们放在一起的话，用一飞行行为的接口，比如命名为IFly的接口来处理就是非常好的办法。”

“但是我对抽象类和接口的区别还是不太清楚。”

“问到点子上了，这两个概念的异同点是网上讨论面向对象问题时讨论得最多的话题之一，从表象上来说，抽象类可以给出一些成员的实现，接口却不包含成员的实现，抽象类的抽象成员可被子类部分实现，接口的成员需要实现类完全实现，一个类只能继承一个抽象类，但可实现多个接口等等。但这些都是从两者的形态上去区分的。我觉得还有三点是能帮助我们去区分抽象类和接口的。第一，类是对对象的抽象；抽象类是对类的抽象；接口是对行为的抽象。接口是对类的局部（行为）进行的抽象，而抽象类是对类整体（字段、属性、方法）的抽象。如果只关注行为抽象，那么也可以认为接口就是抽象类。总之，不论是接

口、抽象类、类甚至对象，都是在不同层次、不同角度进行抽象的结果，它们的共性就是抽象。第二，如果行为跨越不同类的对象，可使用接口；对于一些相似的类对象，用继承抽象类。比如猫呀狗呀它们其实都是动物，它们之间有很多相似的地方，所以我们应该让它们去继承动物这个抽象类，而飞机、麻雀、超人是完全不相关的类，叮是动漫角色，孙悟空是古代神话人物，这也是不相关的类，但它们又是有共同点的，前三个都会飞，而后两个都会变出东西，所以此时让它们去实现相同的接口来达到我们的设计目的就很合适了。”

“哦，明白，其实实现接口和继承抽象类并不冲突的，我完全可以让超人继承人类，再实现飞行接口，是吗？”

“对，超人除了内裤外穿以外，基本就是一个正常人的样子，让他继承人类是对的，但他本事很大，除了飞天，他还具有刀枪不入、力大无穷等等非常人的能力，而这些能力也可能是其他对象具备的，所以就让超人去实现飞行、力大无穷等行为接口，这就可以让超人和飞机比飞行，和大象比力气了，这就是一个类只能继承一个抽象类，却可以实现多个接口的做法。”

“那还有一点呢？”

“嗯，这一点更加关键，那就是第三，从设计角度讲，抽象类是从子类中发现了公共的东西，泛化出父类，然后子类继承父类，而接口是根本不知子类的存在，方法如何实现还不确认，预先定义。这里其实说明的是抽象类和接口设计的思维过程。回想一下我们今天刚开始讲的时候，先是有一个Cat类，然后再有一个Dog类，观察后发现它们有类似之处，于是泛化出Animal类，这也体现了敏捷开发的思想，通过重构改善既有代码的设计。事实上，只有小猫的时候，你就去设计一个动物类，这就极有可能会成为过度设计了。所以说抽象类往往都是通过重构得来的，当然，如果你事先意识到多种分类的可能，那么事先就设计出抽象类也是完全可以的。而接口就完全不是一回事，比如我们是动物运动会的主办方，在策划时，大家就坐在一起考虑需要组织什么样的比赛，大家商议后，觉得应该设置如跑得最快、跳得最高、飞得最远、叫得最响、力气最大等等比赛项目，而此时，主办方其实还不太清楚会有什么样的动物来参加运动会，所有的这些比赛项目都可能是完全不相同的动物在比，它们将如何去实现这些行为也不得而知，此时，能做的事就是事先定义这些比赛项目的行为接口。”

“啊，你的意思是不是说，抽象类是自底而上抽象出来的，而接口则是自顶向下设计出来的。”

“对，可以这么说。其实仅仅理解这一点是不够的，要想真正把抽象类和接口用好，还是需要好好用心地去学习设计模式。只有真正把设计模式理解好了，那么你才能算是真正会合理应用抽象类和接口了。”

A.12 集合

“下面我们再来看看，客户端的代码中，‘动物报名’用的是Animal类的对象数组，你设置了数组的长度为5，也就是说最多只能有五个动物可以报名参加‘叫声比赛’，多了就不行了。这显然是非常不合理的，应该考虑改进。你能说说数组的优缺点吗？”

“数组优点，比如说数组在内存中连续存储，因此可以快速而容易地从头到尾遍历元素，可以快速修改元素等等。缺点嘛，应该是创建时必须要指定数组变量的大小，还有在两个元素之间添加元素也比较困难。”

“说得不错，的确是这样，这就可能使得数组长度设置过大，造成内存空间浪费，长度设置过小造成溢出。所以**.NET Framework**提供了用于数据存储和检索的专用类，这些类统称集合。这些类提供对堆栈、队列、列表和哈希表的支持。大多数集合类实现相同的接口。我们现在介绍当中最常用的一种，ArrayList。”

“集合？它和数组有什么区别？”

“别急，首先**ArrayList**是命名空间**System.Collections**下的一部分，它是使用大小可按需动态增加的数组实现**IList**接口[MSDN]。”

“哦，没学接口前不太懂，现在知道了，你的意思是说，IList接口定义了很多集合用的方法，ArrayList对这些方法做了具体的实现？”

“对的，**ArrayList**的容量是**ArrayList**可以保存的元素数。**ArrayList**的默认初始容量为**0**。随着元素添加到**ArrayList**中，容量会根据需要通过重新分配自动增加。使用整数索引可以访问此集合中的元素。此集合中的索引从零开始。[MSDN]”

“是不是可以这样理解，数组的容量是固定的，而 ArrayList的容量可根据需要自动扩充。”

“是的，由于实现了IList，所以ArrayList提供添加、插入或移除某一范围元素的方法。下面我们来看看如何做。”

```
using System.Collections;          // 增加此命名空间

public partial class Form1 : Form
{
    IList arrayAnimal;             // 声明一集合变量，可以用接口 IList，也可以直接声明“ArrayList arrayAnimal;”

    //动物报名按钮事件
    private void button3_Click(object sender, EventArgs e)
    {
```

```
        arrayAnimal = new ArrayList();

        arrayAnimal.Add(new Cat("小花"));
        arrayAnimal.Add(new Dog("阿毛"));
        arrayAnimal.Add(new Dog("小黑"));
        arrayAnimal.Add(new Cat("娇娇"));
        arrayAnimal.Add(new Cat("咪咪"));

        MessageBox.Show(arrayAnimal.Count.ToString());
    }
}
```

实例化 ArrayList 对象，注意，此时并没有指定 arrayAnimal 的大小，这与数组并不相同

调用集合的 Add 方法增加对象，其参数是 object，所以 new Cat 和 new Dog 都没有问题

随着 Add 对象的增加，集合的 Count 可以得到当前的元素个数

“对于‘叫声比赛’的代码，没有什么变化。”

```
private void button4_Click(object sender, EventArgs e)
{
    foreach (Animal item in arrayAnimal)
    {
        MessageBox.Show(item.Shout());
    }
}
```

此时遍历的是 ArrayList 集合，而非数组

“如果有动物报完名后，由于某种原因（比如政治、宗教、兴奋剂、健康等等）放弃比赛，此时应该需要将其从名单中移除。例如，在报了名后，两只小狗需要退出比赛。我们查了一下它们的报名索引次序为1和2（从0开始计算），所以我们可以应用集合的RemoveAt方法，它的作用是移除指定索引处的集合项。”

“我明白怎么做了。”

```
private void button3_Click(object sender, EventArgs e)
{
    arrayAnimal = new ArrayList();
    arrayAnimal.Add(new Cat("小花"));
    arrayAnimal.Add(new Dog("阿毛"));
    arrayAnimal.Add(new Dog("小黑"));
    arrayAnimal.Add(new Cat("娇娇"));
    arrayAnimal.Add(new Cat("咪咪"));

    arrayAnimal.RemoveAt(1);
    arrayAnimal.RemoveAt(2);
}
```

阿毛和小黑要退出比赛，所以移除他们，但这样处理不能满足需求

“哈，你太着急，集合与数组的不同就在于此，程序在执行RemoveAt（1）的时候，也就是叫‘阿毛’的Dog被移除了集合，此时‘小黑’的索引次序还是原来的2吗？”

“哦，我明白了，等于整个后序对象都向前移一位了。应该是这样才对。也就是说，集合的变化是影响全局的，它始终都保证元素的连续性。”

```
arrayAnimal.RemoveAt(1);
arrayAnimal.RemoveAt(1);
```

“总结一下，集合ArrayList相比数组有什么好处？”

“主要就是它可以根据使用大小按需动态增加，不用受事先设置其大小的控制。还有就是可以随意地添加、插入或移除某一范围元素，比数组要方便。”

“对，这是ArrayList的优势，但它也有不足，ArrayList不管你是什么对象都是接受的，因为在它眼里所有元素都是Object，这就使得如果你‘arrayAnimal.Add（123）;’或者‘arrayAnimal.Add（“HelloWorld”）;’在编译时都是没有问题的，但在执行时，‘foreach（Animal item in arrayAnimal）’需要明确集合中的元素是Animal类型，而123是整型，HelloWorld是字符串型，这就会在运行到此处时报错，显然，这是典型的类型不匹配错误，换句话说，**ArrayList**不是类型安全的。还有就是ArrayList对于存放值类型的数据，比如int、string型（string是一种拥有值类型特点的特殊引用类型）或者结构struct的数据，用ArrayList就意味着都需要将值类型装箱为Object对象，使用集合元素时，还需要执行拆箱操作，这就带来了很大的性能损耗。”

“等等，我不太懂，装箱和拆箱是什么意思？”

“所谓装箱就是把值类型打包到**Object**引用类型的一个实例中。比如整型变量i被“装箱”并赋值给对象o。”

```
int i = 123;
object o =(object) i;  // boxing
```

“所谓拆箱就是指从对象中提取值类型。此例中对象o拆箱并将其赋值给整型变量i”

```
o = 123;
i =(int) o;  // unboxing
```

“相对于简单的赋值而言，装箱和拆箱过程需要进行大量的计算。对

值类型进行装箱时，必须分配并构造一个全新的对象。其次，拆箱所需的强制转换也需要进行大量的计算[MSDN]。总之，装箱拆箱是耗资源和时间的。而ArrayList集合在使用值类型数据时，其实就是在不断地做装箱和拆箱的工作，这显然是非常糟糕的事情。”

“啊，那从这点上来看，它还不如数组来得好了，因为数组事先就指定了数据类型，就不会有类型安全的问题，也不存在装箱和拆箱的事情了。看来他们各有利弊呀。”

“说得非常对，C#在2.0版之前的确也没什么好办法，但2.0出来后，就推出了新的技术来解决这个问题，那就是泛型。”

A.13 泛型

“泛型是具有占位符（类型参数）的类、结构、接口和方法，这些占位符是类、结构、接口和方法所存储或使用的一个或多个类型的占位符。泛型集合类可以将类型参数用作它所存储的对象的类型的占位符；类型参数作为其字段的类型和其方法的参数类型出现[MSDN]。我读给你的是MSDN的原话，听起有些抽象，我们直接来看例子。首先泛型集合需要System.Collections.Generic的命名空间。而List类是ArrayList类的泛型等效类。该类使用大小可按需动态增加的数组实现IList泛型接口。其实用法上关键就在IList和List后面加‘＜T＞’，这个‘T’就是你需要指定的集合的数据或对象类型。”

```
using System.Collections.Generic;

public partial class Form1 : Form
{
    IList<Animal> arrayAnimal;

    //动物报名按钮事件
    private void button3_Click(object sender, EventArgs e)
    {

        arrayAnimal = new List<Animal>();

        arrayAnimal.Add(new Cat("小花"));
        arrayAnimal.Add(new Dog("阿毛"));
        arrayAnimal.Add(new Dog("小黑"));
        arrayAnimal.Add(new Cat("娇娇"));
        arrayAnimal.Add(new Cat("咪咪"));

        MessageBox.Show(arrayAnimal.Count.ToString());
    }
}
```

增加泛型集合命名空间

关键就在这里，声明一泛型集合变量，用接口 IList，注意：IList<Animal>表示此集合变量只能接受 Animal 类型，其他不可以。也可以直接声明“List <Animal> arrayAnimal;”

实例化 List 对象，注意，此时也需要指定 List<T>的‘T’是 Animal

“此时，如果你再写‘arrayAnimal.Add（123）;’或者‘arrayAnimal.Add（“HelloWorld”）;’结果将是？”

“哈，编译就报错，因为Add的参数必须是要Animal或者Animal的子类型才行。”

“至于‘叫声比赛’的写法就完全相同，这里就不写了。来，说说你对泛型集合List的感受。”

“我是这样想的，其实List和ArrayList在功能上是一样的，不同就在于，它在声明和实例化时都需要指定其内部项的数据或对象类型，这就避免了刚才讲的类型安全问题和装箱拆箱的性能问题了。强，够强，怎么想到的，真是厉害。”

“是呀，这个改造的确是非常的好，不过显然C#语言的设计者也并

不是一开始就明白这一点，也是通过实践和用户的反馈才在C#2.0版中改进过来的。巨人也有会走弯路的时候，何况我们常人。通常情况下，都建议使用泛型集合，因为这样可以获得类型安全的直接优点而不需要从基集合类型派生并实现类型特定的成员。此外，如果集合元素为值类型，泛型集合类型的性能通常优于对应的非泛型集合类型（并优于从非泛型基集合类型派生的类型），因为使用泛型时不必对元素进行装箱[MSDN]。”

“当然是泛型好呀，它可是集ArrayList集合和Array数组优点于一身的好东西，有了它，ArrayList就显得太老土了。”

“至于泛型的知识还有很多，这里就不细讲了，你自己去找资料研究吧。”

“好的好的，其实已经有些明白是怎么回事了。我自己去研究吧。”

A.14　委托与事件

“蔡老师，能不能给我讲讲委托与事件，我认真地看过书，也研究过，但实话说，我至今也不是很明白它的作用和好处。”

“哈，委托是对函数的封装，可以当作给方法的特征指定一个名称。而事件则是委托的一种特殊形式，当发生有意义的事情时，事件对象处理通知过程[PC#]。事件其实就是设计模式中观察者模式在.NET中的一种实现方式。要详细讲就有得讲了，不过在这可以举个例子，就当抛砖引玉吧。”

“好的，好的。”

“你先建一个控制台应用程序。我们的需求是，有一只猫叫Tom，有两只老鼠叫Jerry和Jack，Tom只要一叫‘喵，我是Tom’，两只老鼠就说‘老猫来了，快跑’。你来分析一下，这里应该有几个类，如何处理类之间的关系？”

“当然应该是有Cat和Mouse类，在Main函数中执行，当Cat的Shout方法触发时，Mouse就执行Run方法。不过这里如何让Shout触发时，通知两只老鼠呢？显然老猫不会认识老鼠，也不会主动通知它们‘我来了，你们快跑’。”

“你说得没错，的确是这样。所以在Cat类当中，是不应该关联Mouse类的。此时用委托事件的方式就是最好的处理办法了。注意，委托是一种引用方法的类型。一旦为委托分配了方法，委托将与该方法具有完全相同的行为[MSDN]。委托对象用关键字delegate来声明。而事件是说在发生其他类或对象关注的事情时，类或对象可通过事件通知它们[MSDN]。事件对象用event关键字声明。”

```
public delegate void CatShoutEventHandler();
```

```
public event CatShoutEventHandler CatShout;
```

“这里就是声明了一个委托，显然委托名称叫做CatShoutEventHandler，而这个委托所能代表的方法是无参数、无返回值的方法。然后声明了一个对外公开的public事件CatShout，它的事件类型是委托CatShoutEventHandler。表明事件发生时，执行被委托的方法。关键的代码解释后，我们来看实际代码。”

```
class Cat
{
    private string name;
    public Cat(string name)
    {
        this.name = name;
    }

    public delegate void CatShoutEventHandler();

    public event CatShoutEventHandler CatShout;

    public void Shout()
    {
        Console.WriteLine("喵,我是{0}.", name);

        if (CatShout != null)
        {
            CatShout();
        }
    }
}
```

声明委托 CatShoutEventHandler

声明事件 CatShout，它的事件类型是委托 CatShoutEventHandler

表明当执行 Shout()方法时，如果 CatShout 中有对象登记事件，则执行 CatShout()

“我问你，为什么CatShout（ ）是无参数、无返回值的方法？”

“因为事件CatShout的类型是委托CatShoutEventHandler，而CatShoutEventHandler就是无参数、无返回值的。”

“说得对，就这么简单。我们再来看Mouse，它更加简单。”

```
class Mouse
{
    private string name;
    public Mouse(string name)
    {
        this.name = name;
    }

    public void Run()
    {
        Console.WriteLine("老猫来了，{0}快跑！", name);
    }
}
```

用来逃跑的方法

“关键是Main函数的写法。”

```
static void Main(string[] args)
{
    Cat cat = new Cat("Tom");
    Mouse mouse1 = new Mouse("Jerry");
    Mouse mouse2 = new Mouse("Jack");

    cat.CatShout += new Cat.CatShoutEventHandler(mouse1.Run);
    cat.CatShout += new Cat.CatShoutEventHandler(mouse2.Run);

    cat.Shout();

    Console.Read();
}
```

实例化老猫 Tom 以及小老鼠 Jerry 和 Jack

表示将 Mouse 的 Run 方法通过实例化委托 Cat.CatShoutEventHandler 登记到 Cat 的事件 CatShout 当中。其中"+="表示"add_CatShout"的意思

"这里需要解释一下，new Cat.CatShoutEventHandler（mouse1.Run）的含义是实例化一个委托，而委托的实例其实就是Mouse的Run方法。而'cat.CatShout ＋＝'表示的就是'cat.add_CatShout（new Cat.CatShoutEventHandler（mouse1.Run））的意思。'"

"哦，原来'＋＝'就是增加委托实例对象的意思，以前我一直不明白这里的含义。那要是这么说，应该也有'-＝'，移除委托实例的操作了？"

"当然当然，所谓'-＝'不过就是'remove_CatShout（ ）'的含义了，使用了它，就等于减少一个需要触发事件时通知的对象。我们来看运行后的结果。"

```
喵，我是Tom。
老猫来了，Jerry快跑！
老猫来了，Jack快跑！
```

"我还有个问题，我看到我在写.NET应用程序或者Web程序时，总是看到IDE生成的事件参数。比如private void button1_Click（object sender，EventArgs e），这里的sender和e有什么用呀？"

"问得好，我们来改造一下这个例子，你就知道这两个参数是做什么用的了。"

"首先我们增加一个类CatShoutEventArgs，让它继承EventArgs，**EventArgs**是包含事件数据的类的基类[MSDN]。换句话说，这个类的作用就是用来在事件触发时传递数据用的。我现在写了它的一个子类CatShoutEventArgs，当中有属性Name表示的就是CatShout事件触发时，需要传递Cat对象的名字。"

```
public class CatShoutEventArgs : EventArgs
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

继承 EventArgs

“然后改写Cat类的代码，对委托CatShoutEventHandler进行重定义。增加两个参数，第一个参数object对象sender是指向发送通知的对象，而第二个参数CatShoutEventArgs的args，包含了所有通知接受者需要附件的信息。在这里显然就是老猫的名字信息。”

```
class Cat
{
    private string name;
    public Cat(string name)
    {
        this.name = name;
    }

    public delegate void CatShoutEventHandler(object sender, CatShoutEventArgs args);
    public event CatShoutEventHandler CatShout;

    public void Shout()
    {
        Console.WriteLine("喵,我是{0}.", name);

        if (CatShout != null)
        {
            CatShoutEventArgs e = new CatShoutEventArgs();
            e.Name = this.name;
            CatShout(this, e);
        }
    }
}
```

声明委托 CatShoutEventHandler，此时委托所代表的方法有两个参数，object 参数和 CatShoutEventArgs 参数

声明并实例化一个 CatShoutEventArgs，并给 Name 属性赋值为猫的名字

当事件触发时，通知所有登记过的对象，并将发送通知的自己以及需要的数据传递过去

“哦，原来object　sender就是传递发送通知的对象，而EventArgs是包含事件数据的类，怪不得。”

“Mouse类也发生了一点小变化。由于有了传递过来的猫的名字，所以显示的时候可以指示是老猫谁谁谁来了。”

```
class Mouse
{
    private string name;
    public Mouse(string name)
    {
        this.name = name;
    }

    public void Run(object sender, CatShoutEventArgs args)
    {
        Console.WriteLine("老猫{0}来了，{1}快跑！", args.Name, name);
    }
}
```

逃跑的方法中增加了两个参数，并且可以在显示时，说出老猫的名字

“Main函数的代码没有变化，而结果显示不一样了。”

```
喵，我是Tom。
老猫Tom来了，Jerry快跑！
老猫Tom来了，Jack快跑！
```

“啊，原来委托事件这么简单。看来我以前是没认真学。那蔡老师，我们下面学什么？”

A.15 客套

“要讲的东西太多了，我们的‘动物运动会’程序也只写了个开头，以后有的是机会。”小菜看了看表说，“现在都过了中午，食堂都快没菜了，走，我们先吃饭去吧。”

“好的，今天真的太感谢了，我觉得这半天的收获远远比上一个月课，看几本砖头书来得效果好呀。”史熙兴奋地说。

“哪里哪里，今天讲的都只是皮毛，要学习的内容还多着呢，不过话说回来，上午讲的这个未完成的‘动物运动会’的例子尽管简单，但却涵盖了面向对象的最重要的知识，你好好去体会一下。有机会我再跟你讲讲设计模式，你对封装、继承、多态的理解就会更深入一些，学无止境，你需要不断地练习实践才可能真正成为优秀的软件工程师。”

“嗯，我觉得我对编程有了很大的兴趣，面向对象的编程方式确实非常有意思。”

“师傅领进门，修行在个人，今后就看你的了，好好加油。不过现在我们还是先去为肚皮加点油哦。”

“对对对，走，我们去吃饭去，先去加点食物油。”

附录B 参考文献

➢ **[DP]** 中文书名：《设计模式：可复用面向对象软件的基础》作者：Erich Gamm，Richard Helm，Ralph Johnson，John Vlissides，简称：GoF，译者：李英军、马晓星、蔡敏、刘建中，机械工业出版社

➢ http://www.dofactory.com/Patterns/Patterns.aspx网站名：**data & object factory**本书的设计模式基本结构图和基本代码都来自此网站

➢ **[DPE]** 中文书名：《设计模式解析》（第2版）作者：Alan Shalloway，James R. Trott，译者：徐言声，人民邮电出版社

➢ **[ASD]** 中文书名：《敏捷软件开发：原则、模式与实践》作者：Robert C.Martin，译者：邓辉，清华大学出版社

➢ **[RIDEC]** 中文书名：《重构——改善既有代码的设计》作者：Martin Fowler，译者：侯捷、熊节，中国电力出版社

➢ **[J&DP]** 中文书名：《**Java**与模式》作者：阎宏，电子工业出版社

➢ **[R2P]** 中文书名：《重构与模式》作者：Joshua Kerievsky，译者：杨光、刘基诚，人民邮电出版社

➢ 《**Head First Design Patterns**》（影印版）作者：Eric Freeman & Elisabeth Freeman with Kathy Sierra & Bert Bates，东南大学出版社

➢ MSDN WebCast 《C#面向对象设计模式纵横谈》讲师：李建忠

http://www.msdnwebcast.com.cn/CourseSeries.aspx?id＝12

➢ 《企业应用架构模式》作者：Martin Fowler，译者：王怀民、周斌，机械工业出版社

➢ **[AMNFP]** 《**Microsoft .NET**框架程序设计》（修订版）作者：Jeffrey Richter，译者：李建忠，清华大学出版社

➢ 《面向对象分析与设计》（原书第2版） 作者：GRADY BOOCH，译者：冯博琴、冯岚、薛涛、崔舒宁，机械工业出版社

- 《.Net设计模式》作者：甄镭，电子工业出版社
- 《深入浅出设计模式 C#/Java版》作者：莫勇腾，清华大学出版社
- 《C#入门经典》作者：KARLI WATSON，CHRISTIAN NAGEL，译者：齐立波，清华大学出 版社
- 《C#设计模式》作者：STEVE JOHN METSKER，译者：颜炯，中国电力出版社
- [PC#]《C#高级编程》（第3版）作者：CHRISTIAN NAGEL，BILL EVJEN，JAY GLYNN，译者：李敏波，清华大学出版社
- 《编程的奥秘——.Net软件技术学习与实践》作者：金旭亮，电子工业出版社
- http://wayfarer.cnblogs.com/博客作者：张逸
- http://zhenyulu.cnblogs.com/博客作者：吕震宇
- http://terrylee.cnblogs.com/博客作者：李会军
- http://idior.cnblogs.com/ 博客作者：idior
- http://allenlooplee.cnblogs.com/ 博客作者：Allen Lee
- http://www.jdon.com/designpatterns/index.htm网站名：J道，版主：banq
- http://blog.csdn.net/billdavid/category/22838.aspx 博客作者：大卫
- http://blog.csdn.net/ai92/ 博客作者：ai92

Table of Contents

扉页
目录
版权页
内容简介
序
本书出版后的读者评论
前言
第1章 代码无错就是优？——简单工厂模式
- 1.1 面试受挫
- 1.2 初学者代码毛病
- 1.3 代码规范
- 1.4 面向对象编程
- 1.5 活字印刷，面向对象
- 1.6 面向对象的好处
- 1.7 复制vs.复用
- 1.8 业务的封装
- 1.9 紧耦合vs.松耦合
- 1.10 简单工厂模式
- 1.11 UML类图

第2章 商场促销——策略模式
- 2.1 商场收银软件
- 2.2 增加打折
- 2.3 简单工厂实现
- 2.4 策略模式
- 2.5 策略模式实现
- 2.6 策略与简单工厂结合
- 2.7 策略模式解析

第3章 拍摄UFO——单一职责原则
- 3.1 新手机
- 3.2 拍摄
- 3.3 没用的东西
- 3.4 单一职责原则
- 3.5 方块游戏的设计
- 3.6 手机职责过多吗？

第4章 考研求职两不误——开放-封闭原则
- 4.1 考研失败
- 4.2 开放-封闭原则
- 4.3 何时应对变化
- 4.4 两手准备，并全力以赴

第5章 会修电脑不会修收音机？——依赖倒转原则
- 5.1 MM请求修电脑

5.2 电话遥控修电脑
5.3 依赖倒转原则
5.4 里氏代换原则
5.5 修收音机
第6章 穿什么有这么重要？——装饰模式
6.1 穿什么有这么重要？
6.2 小菜扮靓第一版
6.3 小菜扮靓第二版
6.4 装饰模式
6.5 小菜扮靓第三版
6.6 装饰模式总结
第7章 为别人做嫁衣——代理模式
7.1 为别人做嫁衣！
7.2 没有代理的代码
7.3 只有代理的代码
7.4 符合实际的代码
7.5 代理模式
7.6 代理模式应用
7.7 秀才让小六代其求婚
第8章 雷锋依然在人间——工厂方法模式
8.1 再现活雷锋
8.2 简单工厂模式实现
8.3 工厂方法模式实现
8.4 简单工厂vs.工厂方法
8.5 雷锋工厂
第9章 简历复印——原型模式
9.1 夸张的简历
9.2 简历代码初步实现
9.3 原型模式
9.4 简历的原型实现
9.5 浅复制与深复制
9.6 简历的深复制实现
9.7 复制简历vs.手写求职信
第10章 考题抄错会做也白搭——模板方法模式
10.1 选择题不会做，蒙呗！
10.2 重复＝易错＋难改
10.3 提炼代码
10.4 模板方法模式
10.5 模板方法模式特点
10.6 主观题，看你怎么蒙
第11章 无熟人难办事？——迪米特法则
11.1 第一天上班
11.2 无熟人难办事
11.3 迪米特法则

第12章 牛市股票还会亏钱？——外观模式
12.1 牛市股票还会亏钱？
12.2 股民炒股代码
12.3 投资基金代码
12.4 外观模式
12.5 何时使用外观模式
第13章 好菜每回味不同——建造者模式
13.1 炒面没放盐
13.2 建造小人一
13.3 建造小人二
13.4 建造者模式
13.5 建造者模式解析
13.6 建造者模式基本代码
第14章 老板回来，我不知道——观察者模式
14.1 老板回来？我不知道！
14.2 双向耦合的代码
14.3 解耦实践一
14.4 解耦实践二
14.5 观察者模式
14.6 观察者模式特点
14.7 观察者模式的不足
14.8 事件委托实现
14.9 事件委托说明
14.10 石守吉失手机后的委托
第15章 就不能不换DB吗？——抽象工厂模式
15.1 就不能不换DB吗？
15.2 最基本的数据访问程序
15.3 用了工厂方法模式的数据访问程序
15.4 用了抽象工厂模式的数据访问程序
15.5 抽象工厂模式
15.6 抽象工厂模式的优点与缺点
15.7 用简单工厂来改进抽象工厂
15.8 用反射＋抽象工厂的数据访问程序
15.9 用反射＋配置文件实现数据访问程序
15.10 无痴迷，不成功
第16章 无尽加班何时休——状态模式
16.1 加班，又是加班！
16.2 工作状态-函数版
16.3 工作状态-分类版
16.4 方法过长是坏味道
16.5 状态模式
16.6 状态模式好处与用处
16.7 工作状态-状态模式版
第17章 在NBA我需要翻译——适配器模式

17.1 在NBA我需要翻译！
17.2 适配器模式
17.3 何时使用适配器模式
17.4 篮球翻译适配器
17.5 适配器模式的.NET应用
17.6 扁鹊的医术
第18章 如果再回到从前——备忘录模式
18.1 如果再给我一次机会……
18.2 游戏存进度
18.3 备忘录模式
18.4 备忘录模式基本代码
18.5 游戏进度备忘
第19章 分公司＝一部门——组合模式
19.1 分公司不就是一部门吗？
19.2 组合模式
19.3 透明方式与安全方式
19.4 何时使用组合模式
19.5 公司管理系统
19.6 组合模式好处
第20章 想走？可以！先买票——迭代器模式
20.1 乘车买票，不管你是谁！
20.2 迭代器模式
20.3 迭代器实现
20.4 .NET的迭代器实现
20.5 迭代高手
第21章 有些类也需计划生育——单例模式
21.1 类也需要计划生育
21.2 判断对象是否是null
21.3 生还是不生是自己的责任
21.4 单例模式
21.5 多线程时的单例
21.6 双重锁定
21.7 静态初始化
第22章 手机软件何时统一——桥接模式
22.1 凭什么你的游戏我不能玩
22.2 紧耦合的程序演化
22.3 合成/聚合复用原则
22.4 松耦合的程序
22.5 桥接模式
22.6 桥接模式基本代码
22.7 我要开发“好”游戏
第23章 烤羊肉串引来的思考——命令模式
23.1 吃烤羊肉串！
23.2 烧烤摊vs.烧烤店

23.3 紧耦合设计
23.4 松耦合设计
23.5 松耦合后
23.6 命令模式
23.7 命令模式作用
第24章 加薪非要老总批？——职责链模式
24.1 老板，我要加薪！
24.2 加薪代码初步
24.3 职责链模式
24.4 职责链的好处
24.5 加薪代码重构
24.6 加薪成功
第25章 世界需要和平——中介者模式
25.1 世界需要和平！
25.2 中介者模式
25.3 安理会做中介
25.4 中介者模式优缺点
第26章 项目多也别傻做——享元模式
26.1 项目多也别傻做！
26.2 享元模式
26.3 网站共享代码
26.4 内部状态与外部状态
26.5 享元模式应用
第27章 其实你不懂老板的心——解释器模式
27.1 其实你不懂老板的心
27.2 解释器模式
27.3 解释器模式好处
27.4 音乐解释器
27.5 音乐解释器实现
27.6 料事如神
第28章 男人和女人——访问者模式
28.1 男人和女人！
28.2 最简单的编程实现
28.3 简单的面向对象实现
28.4 用了模式的实现
28.5 访问者模式
28.6 访问者模式基本代码
28.7 比上不足，比下有余
第29章 OOTV杯超级模式大赛——模式总结
29.1 演讲任务
29.2 报名参赛
29.3 超模大赛开幕式
29.4 创建型模式比赛
29.5 结构型模式比赛

29.6 行为型模式一组比赛
29.7 行为型模式二组比赛
29.8 决赛
29.9 梦醒时分
29.10 没有结束的结尾
附录A 培训实习生——面向对象基础
A.1 培训实习生
A.2 类与实例
A.3 构造方法
A.4 方法重载
A.5 属性与修饰符
A.6 封装
A.7 继承
A.8 多态
A.9 重构
A.10 抽象类
A.11 接口
A.12 集合
A.13 泛型
A.14 委托与事件
A.15 客套
附录B 参考文献